Quantitative Economics with Python

Thomas J. Sargent and John Stachurski

March 3, 2020

Contents

I Introduction to Python 1

1 About Python 3

2 Setting up Your Python Environment 15

3 An Introductory Example 33

4 Functions 51

5 Python Essentials 61

6 OOP I: Introduction to Object Oriented Programming 81

7 OOP II: Building Classes 87

II The Scientific Libraries 105

8 Python for Scientific Computing 107

9 NumPy 117

10 Matplotlib 137

11 SciPy 149

12 Numba 159

13 Parallelization 173

14 Pandas 183

III Advanced Python Programming 199

15 Writing Good Code 201

16 More Language Features 215

17 Debugging 255

IV Tools and Techniques 263

18 Geometric Series for Elementary Economics 265

19 Linear Algebra 285

20 Complex Numbers and Trigonometry 309

21 Orthogonal Projections and Their Applications 319

22 LLN and CLT 335

23 Heavy-Tailed Distributions 353

V Introduction to Dynamics 371

24 Dynamics in One Dimension 373

25 AR1 Processes 393

26 Finite Markov Chains 407

27 Inventory Dynamics 431

28 Linear State Space Models 441

29 Application: The Samuelson Multiplier-Accelerator 465

30 Kesten Processes and Firm Dynamics 501

31 Wealth Distribution Dynamics 515

32 Continuous State Markov Chains 531

33 Cass-Koopmans Optimal Growth Model 553

34 A First Look at the Kalman Filter 583

35 Reverse Engineering a la Muth 601

VI Dynamic Programming 609

36 Shortest Paths 611

37 Job Search I: The McCall Search Model 621

38 Job Search II: Search and Separation 637

39 Job Search III: Fitted Value Function Iteration 651

40 Job Search IV: Correlated Wage Offers 661

41 Exchangeability and Bayesian Updating 671

42 Job Search V: Search with Learning 687

43 Job Search VI: Modeling Career Choice 715

44 Job Search VII: On-the-Job Search 729

45 A Problem that Stumped Milton Friedman 741

46 Optimal Growth I: The Stochastic Optimal Growth Model 759

47 Optimal Growth II: Accelerating the Code with Numba 775

48 Optimal Growth III: Time Iteration 785

49 Optimal Growth IV: The Endogenous Grid Method 803

50 The Income Fluctuation Problem 813

51 Discrete State Dynamic Programming 829

VII LQ Control 851

52 LQ Control: Foundations 853

53 Optimal Savings I: The Permanent Income Model 883

54 Optimal Savings II: LQ Techniques 901

55 Information and Consumption Smoothing 919

56 Consumption Smoothing with Complete and Incomplete Markets 935

57 Tax Smoothing with Complete and Incomplete Markets 951

58 Robustness 985

59 Markov Jump Linear Quadratic Dynamic Programming 1005

60 How to Pay for a War: Part 1 1045

61 How to Pay for a War: Part 2 1055

62 How to Pay for a War: Part 3 1071

63 Optimal Taxation in an LQ Economy 1077

64 Production Smoothing via Inventories 1099

VIII Data and Empirics 1119

65 Pandas for Panel Data 1121

66 Linear Regression in Python 1143

67 Maximum Likelihood Estimation 1161

IX Multiple Agent Models 1183

68 Schelling's Segregation Model 1185

69 A Lake Model of Employment and Unemployment 1197

70 Rational Expectations Equilibrium 1223

71 Markov Perfect Equilibrium 1239

72 Robust Markov Perfect Equilibrium 1257

73 Uncertainty Traps 1277

74 The Aiyagari Model 1291

75 Default Risk and Income Fluctuations 1301

76 Globalization and Cycles 1321

77 Coase's Theory of the Firm 1339

X Dynamic Linear Economies 1353

78 Recursive Models of Dynamic Linear Economies 1355

79 Growth in Dynamic Linear Economies 1391

80 Lucas Asset Pricing Using DLE 1403

81 IRFs in Hall Models 1411

82 Permanent Income Model using the DLE Class 1419

83 Rosen Schooling Model 1425

84 Cattle Cycles 1431

85 Shock Non Invertibility 1439

XI Classic Linear Models 1447

86 Von Neumann Growth Model (and a Generalization) 1449

XII Time Series Models 1467

87 Covariance Stationary Processes 1469

88 Estimation of Spectra 1491

89 Additive and Multiplicative Functionals 1505

90 Classical Control with Linear Algebra 1531

91 Classical Prediction and Filtering With Linear Algebra 1553

XIII Asset Pricing and Finance 1575

92 Asset Pricing I: Finite State Models 1577

93 Asset Pricing II: The Lucas Asset Pricing Model **1599**

94 Asset Pricing III: Incomplete Markets **1609**

95 Two Modifications of Mean-Variance Portfolio Theory **1621**

XIV Dynamic Programming Squared **1647**

96 Stackelberg Plans **1649**

97 Ramsey Plans, Time Inconsistency, Sustainable Plans **1675**

98 Optimal Taxation with State-Contingent Debt **1701**

99 Optimal Taxation without State-Contingent Debt **1735**

100 Fluctuating Interest Rates Deliver Fiscal Insurance **1765**

101 Fiscal Risk and Government Debt **1793**

102 Competitive Equilibria of a Model of Chang **1823**

103 Credible Government Policies in a Model of Chang **1855**

Part I

Introduction to Python

Chapter 1

About Python

1.1 Contents

- Overview 1.2
- What's Python? 1.3
- Scientific Programming 1.4
- Learn More 1.5

> "Python has gotten sufficiently weapons grade that we don't descend into R anymore. Sorry, R people. I used to be one of you but we no longer descend into R."
> – Chris Wiggins

1.2 Overview

In this lecture we will

- outline what Python is
- showcase some of its abilities
- compare it to some other languages.

At this stage, it's **not** our intention that you try to replicate all you see.

We will work through what follows at a slow pace later in the lecture series.

Our only objective for this lecture is to give you some feel of what Python is, and what it can do.

1.3 What's Python?

Python is a general-purpose programming language conceived in 1989 by Dutch programmer Guido van Rossum.

Python is free and open source, with development coordinated through the Python Software Foundation.

Python has experienced rapid adoption in the last decade and is now one of the most popular programming languages.

1.3.1 Common Uses

Python is a general-purpose language used in almost all application domains such as

- communications
- web development
- CGI and graphical user interfaces
- game development
- multimedia, data processing, security, etc., etc., etc.

Used extensively by Internet services and high tech companies including

- Google
- Dropbox
- Reddit
- YouTube
- Walt Disney Animation.

Python is very beginner-friendly and is often used to teach computer science and programming.

For reasons we will discuss, Python is particularly popular within the scientific community with users including NASA, CERN and practically all branches of academia.

It is also replacing familiar tools like Excel in the fields of finance and banking.

1.3.2 Relative Popularity

The following chart, produced using Stack Overflow Trends, shows one measure of the relative popularity of Python

The figure indicates not only that Python is widely used but also that adoption of Python has accelerated significantly since 2012.

We suspect this is driven at least in part by uptake in the scientific domain, particularly in

rapidly growing fields like data science.

For example, the popularity of pandas, a library for data analysis with Python has exploded, as seen here.

(The corresponding time path for MATLAB is shown for comparison)

Note that pandas takes off in 2012, which is the same year that we see Python's popularity begin to spike in the first figure.

Overall, it's clear that

- Python is one of the most popular programming languages worldwide.
- Python is a major tool for scientific computing, accounting for a rapidly rising share of scientific work around the globe.

1.3.3 Features

Python is a high-level language suitable for rapid development.

It has a relatively small core language supported by many libraries.

Other features of Python:

- multiple programming styles are supported (procedural, object-oriented, functional, etc.)
- it is interpreted rather than compiled.

1.3.4 Syntax and Design

One nice feature of Python is its elegant syntax — we'll see many examples later on.

Elegant code might sound superfluous but in fact it's highly beneficial because it makes the syntax easy to read and easy to remember.

Remembering how to read from files, sort dictionaries and other such routine tasks means that you don't need to break your flow in order to hunt down correct syntax.

Closely related to elegant syntax is an elegant design.

Features like iterators, generators, decorators and list comprehensions make Python highly expressive, allowing you to get more done with less code.

Namespaces improve productivity by cutting down on bugs and syntax errors.

1.4 Scientific Programming

Python has become one of the core languages of scientific computing.

It's either the dominant player or a major player in

- machine learning and data science
- astronomy
- artificial intelligence
- chemistry
- computational biology
- meteorology

Its popularity in economics is also beginning to rise.

This section briefly showcases some examples of Python for scientific programming.

- All of these topics will be covered in detail later on.

1.4.1 Numerical Programming

Fundamental matrix and array processing capabilities are provided by the excellent NumPy library.

NumPy provides the basic array data type plus some simple processing operations.

For example, let's build some arrays

```
In [1]: import numpy as np                          # Load the library

        a = np.linspace(-np.pi, np.pi, 100)         # Create even grid from -π to π
        b = np.cos(a)                               # Apply cosine to each element of a
        c = np.sin(a)                               # Apply sin to each element of a
```

Now let's take the inner product

```
In [2]: b @ c

Out[2]: 2.706168622523819e-16
```

The number you see here might vary slightly but it's essentially zero.

(For older versions of Python and NumPy you need to use the np.dot function)

The SciPy library is built on top of NumPy and provides additional functionality.

For example, let's calculate $\int_{-2}^{2} \phi(z)dz$ where ϕ is the standard normal density.

In [3]:
```
from scipy.stats import norm
from scipy.integrate import quad

ϕ = norm()
value, error = quad(ϕ.pdf, -2, 2)  # Integrate using Gaussian quadrature
value
```

Out[3]:
```
0.9544997361036417
```

SciPy includes many of the standard routines used in

- linear algebra
- integration
- interpolation
- optimization
- distributions and random number generation
- signal processing

See them all here.

1.4.2 Graphics

The most popular and comprehensive Python library for creating figures and graphs is Matplotlib, with functionality including

- plots, histograms, contour images, 3D graphs, bar charts etc.
- output in many formats (PDF, PNG, EPS, etc.)
- LaTeX integration

Example 2D plot with embedded LaTeX annotations

Example contour plot

Example 3D plot

More examples can be found in the Matplotlib thumbnail gallery.

Other graphics libraries include

- Plotly
- Bokeh
- VPython — 3D graphics and animations

1.4.3 Symbolic Algebra

It's useful to be able to manipulate symbolic expressions, as in Mathematica or Maple.

The SymPy library provides this functionality from within the Python shell.

In [4]:
```
from sympy import Symbol

x, y = Symbol('x'), Symbol('y')  # Treat 'x' and 'y' as algebraic symbols
x + x + x + y
```

Out[4]: $3x + y$

We can manipulate expressions

In [5]:
```
expression = (x + y)**2
expression.expand()
```

Out[5]: $x^2 + 2xy + y^2$

solve polynomials

In [6]:
```
from sympy import solve

solve(x**2 + x + 2)
```

Out[6]:
```
[-1/2 - sqrt(7)*I/2, -1/2 + sqrt(7)*I/2]
```

and calculate limits, derivatives and integrals

In [7]:
```
from sympy import limit, sin, diff

limit(1 / x, x, 0)
```

Out[7]: ∞

In [8]:
```
limit(sin(x) / x, x, 0)
```

Out[8]: 1

In [9]:
```
diff(sin(x), x)
```

Out[9]: $\cos(x)$

The beauty of importing this functionality into Python is that we are working within a fully fledged programming language.

We can easily create tables of derivatives, generate LaTeX output, add that output to figures and so on.

1.4.4 Statistics

Python's data manipulation and statistics libraries have improved rapidly over the last few years.

Pandas

One of the most popular libraries for working with data is pandas.

Pandas is fast, efficient, flexible and well designed.

Here's a simple example, using some dummy data generated with Numpy's excellent `random` functionality.

In [10]:
```
import pandas as pd
np.random.seed(1234)

data = np.random.randn(5, 2)  # 5x2 matrix of N(0, 1) random draws
dates = pd.date_range('28/12/2010', periods=5)

df = pd.DataFrame(data, columns=('price', 'weight'), index=dates)
print(df)
```

```
                   price    weight
2010-12-28  0.471435 -1.190976
2010-12-29  1.432707 -0.312652
2010-12-30 -0.720589  0.887163
2010-12-31  0.859588 -0.636524
2011-01-01  0.015696 -2.242685
```

In [11]:
```
df.mean()
```

Out[11]:
```
price     0.411768
weight   -0.699135
dtype: float64
```

Other Useful Statistics Libraries

- statsmodels — various statistical routines
- scikit-learn — machine learning in Python (sponsored by Google, among others)
- pyMC — for Bayesian data analysis
- pystan Bayesian analysis based on stan

1.4.5 Networks and Graphs

Python has many libraries for studying graphs.

One well-known example is NetworkX. Its features include, among many other things:

- standard graph algorithms for analyzing networks
- plotting routines

Here's some example code that generates and plots a random graph, with node color determined by shortest path length from a central node.

In [12]:
```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline
np.random.seed(1234)

# Generate a random graph
p = dict((i, (np.random.uniform(0, 1), np.random.uniform(0, 1)))
         for i in range(200))
g = nx.random_geometric_graph(200, 0.12, pos=p)
pos = nx.get_node_attributes(g, 'pos')

# Find node nearest the center point (0.5, 0.5)
dists = [(x - 0.5)**2 + (y - 0.5)**2 for x, y in list(pos.values())]
ncenter = np.argmin(dists)

# Plot graph, coloring by path length from central node
p = nx.single_source_shortest_path_length(g, ncenter)
plt.figure()
nx.draw_networkx_edges(g, pos, alpha=0.4)
nx.draw_networkx_nodes(g,
                       pos,
                       nodelist=list(p.keys()),
                       node_size=120, alpha=0.5,
                       node_color=list(p.values()),
                       cmap=plt.cm.jet_r)
plt.show()
```

```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/networkx/drawing/nx_pylab.py:
↪579:
MatplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.
↪Use
np.iterable instead.
  if not cb.iterable(width):
```

1.4.6 Cloud Computing

Running your Python code on massive servers in the cloud is becoming easier and easier.

A nice example is Anaconda Enterprise.

See also

- Amazon Elastic Compute Cloud
- The Google App Engine (Python, Java, PHP or Go)
- Pythonanywhere
- Sagemath Cloud

1.4.7 Parallel Processing

Apart from the cloud computing options listed above, you might like to consider

- Parallel computing through IPython clusters.
- The Starcluster interface to Amazon's EC2.
- GPU programming through PyCuda, PyOpenCL, Theano or similar.

1.4.8 Other Developments

There are many other interesting developments with scientific programming in Python.

Some representative examples include

- Jupyter — Python in your browser with interactive code cells, embedded images and other useful features.

- Numba — Make Python run at the same speed as native machine code!
- Blaze — a generalization of NumPy.
- PyTables — manage large data sets.
- CVXPY — convex optimization in Python.

1.5 Learn More

- Browse some Python projects on GitHub.
- Read more about Python's history and rise in popularity .
- Have a look at some of the Jupyter notebooks people have shared on various scientific topics.
- Visit the Python Package Index.
- View some of the questions people are asking about Python on Stackoverflow.
- Keep up to date on what's happening in the Python community with the Python subreddit.

Chapter 2

Setting up Your Python Environment

2.1 Contents

- Overview 2.2
- Anaconda 2.3
- Jupyter Notebooks 2.4
- Installing Libraries 2.5
- Working with Python Files 2.6
- Exercises 2.7

2.2 Overview

In this lecture, you will learn how to

1. get a Python environment up and running
2. execute simple Python commands
3. run a sample program
4. install the code libraries that underpin these lectures

2.3 Anaconda

The core Python package is easy to install but *not* what you should choose for these lectures.

These lectures require the entire scientific programming ecosystem, which

- the core installation doesn't provide
- is painful to install one piece at a time.

Hence the best approach for our purposes is to install a Python distribution that contains

1. the core Python language **and**

2. compatible versions of the most popular scientific libraries.

The best such distribution is Anaconda.

Anaconda is

- very popular
- cross-platform
- comprehensive
- completely unrelated to the Nicki Minaj song of the same name

Anaconda also comes with a great package management system to organize your code libraries.

All of what follows assumes that you adopt this recommendation!

2.3.1 Installing Anaconda

To install Anaconda, download the binary and follow the instructions.

Important points:

- Install the latest version!
- If you are asked during the installation process whether you'd like to make Anaconda your default Python installation, say yes.

2.3.2 Updating Anaconda

Anaconda supplies a tool called conda to manage and upgrade your Anaconda packages.

One conda command you should execute regularly is the one that updates the whole Anaconda distribution.

As a practice run, please execute the following

1. Open up a terminal
2. Type `conda update anaconda`

For more information on conda, type conda help in a terminal.

2.4 Jupyter Notebooks

Jupyter notebooks are one of the many possible ways to interact with Python and the scientific libraries.

They use a *browser-based* interface to Python with

- The ability to write and execute Python commands.
- Formatted output in the browser, including tables, figures, animation, etc.
- The option to mix in formatted text and mathematical expressions.

Because of these features, Jupyter is now a major player in the scientific computing ecosystem.

Here's an image showing execution of some code (borrowed from here) in a Jupyter notebook

While Jupyter isn't the only way to code in Python, it's great for when you wish to

- start coding in Python
- test new ideas or interact with small pieces of code
- share or collaborate scientific ideas with students or colleagues

These lectures are designed for executing in Jupyter notebooks.

2.4.1 Starting the Jupyter Notebook

Once you have installed Anaconda, you can start the Jupyter notebook.

Either

- search for Jupyter in your applications menu, or
- open up a terminal and type `jupyter notebook`
  - Windows users should substitute "Anaconda command prompt" for "terminal" in

the previous line.

If you use the second option, you will see something like this

The output tells us the notebook is running at `http://localhost:8888/`

- `localhost` is the name of the local machine
- `8888` refers to port number 8888 on your computer

Thus, the Jupyter kernel is listening for Python commands on port 8888 of our local machine.

Hopefully, your default browser has also opened up with a web page that looks something like this

What you see here is called the Jupyter *dashboard*.

If you look at the URL at the top, it should be `localhost:8888` or similar, matching the message above.

Assuming all this has worked OK, you can now click on `New` at the top right and select `Python 3` or similar.

Here's what shows up on our machine:

The notebook displays an *active cell*, into which you can type Python commands.

2.4.2 Notebook Basics

Let's start with how to edit code and run simple programs.

Running Cells

Notice that, in the previous figure, the cell is surrounded by a green border.

This means that the cell is in *edit mode*.

In this mode, whatever you type will appear in the cell with the flashing cursor.

When you're ready to execute the code in a cell, hit `Shift-Enter` instead of the usual `Enter`.

(Note: There are also menu and button options for running code in a cell that you can find by exploring)

Modal Editing

The next thing to understand about the Jupyter notebook is that it uses a *modal* editing system.

This means that the effect of typing at the keyboard **depends on which mode you are in**.

The two modes are

1. Edit mode

- Indicated by a green border around one cell, plus a blinking cursor
- Whatever you type appears as is in that cell

1. Command mode

- The green border is replaced by a grey (or grey and blue) border
- Keystrokes are interpreted as commands — for example, typing b adds a new cell below the current one

To switch to

- command mode from edit mode, hit the `Esc` key or `Ctrl-M`

- edit mode from command mode, hit `Enter` or click in a cell

The modal behavior of the Jupyter notebook is very efficient when you get used to it.

Inserting Unicode (e.g., Greek Letters)

Python supports unicode, allowing the use of characters such as α and β as names in your code.

In a code cell, try typing `\alpha` and then hitting the tab key on your keyboard.

A Test Program

Let's run a test program.

Here's an arbitrary program we can use: http://matplotlib.org/3.1.1/gallery/pie_and_polar_charts/polar_bar.html.

On that page, you'll see the following code

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Fixing random state for reproducibility
np.random.seed(19680801)

# Compute pie slices
N = 20
θ = np.linspace(0.0, 2 * np.pi, N, endpoint=False)
radii = 10 * np.random.rand(N)
width = np.pi / 4 * np.random.rand(N)
colors = plt.cm.viridis(radii / 10.)

ax = plt.subplot(111, projection='polar')
ax.bar(θ, radii, width=width, bottom=0.0, color=colors, alpha=0.5)

plt.show()
```

Don't worry about the details for now — let's just run it and see what happens.

The easiest way to run this code is to copy and paste it into a cell in the notebook.

Hopefully you will get a similar plot.

2.4.3 Working with the Notebook

Here are a few more tips on working with Jupyter notebooks.

Tab Completion

In the previous program, we executed the line `import numpy as np`

- NumPy is a numerical library we'll work with in depth.

After this import command, functions in NumPy can be accessed with `np.function_name` type syntax.

- For example, try `np.random.randn(3)`.

We can explore these attributes of `np` using the `Tab` key.

For example, here we type `np.ran` and hit Tab

Jupyter offers up the two possible completions, `random` and `rank`.

In this way, the Tab key helps remind you of what's available and also saves you typing.

On-Line Help

To get help on `np.rank`, say, we can execute `np.rank?`.

Documentation appears in a split window of the browser, like so

Clicking on the top right of the lower split closes the on-line help.

Other Content

In addition to executing code, the Jupyter notebook allows you to embed text, equations, figures and even videos in the page.

For example, here we enter a mixture of plain text and LaTeX instead of code

Next we `Esc` to enter command mode and then type `m` to indicate that we are writing Markdown, a mark-up language similar to (but simpler than) LaTeX.

(You can also use your mouse to select `Markdown` from the `Code` drop-down box just below the list of menu items)

Now we `Shift+Enter` to produce this

2.4.4 Sharing Notebooks

Notebook files are just text files structured in JSON and typically ending with `.ipynb`.

You can share them in the usual way that you share files — or by using web services such as nbviewer.

The notebooks you see on that site are **static** html representations.

To run one, download it as an `ipynb` file by clicking on the download icon at the top right.

Save it somewhere, navigate to it from the Jupyter dashboard and then run as discussed above.

2.4.5 QuantEcon Notes

QuantEcon has its own site for sharing Jupyter notebooks related to economics – QuantEcon Notes.

Notebooks submitted to QuantEcon Notes can be shared with a link, and are open to comments and votes by the community.

2.5 Installing Libraries

Most of the libraries we need come in Anaconda.

Other libraries can be installed with `pip`.

One library we'll be using is QuantEcon.py.

You can install QuantEcon.py by starting Jupyter and typing

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

into a cell.

Alternatively, you can type the following into a terminal

```
pip install quantecon
```

More instructions can be found on the library page.

To upgrade to the latest version, which you should do regularly, use

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

Another library we will be using is interpolation.py.

This can be installed by typing in Jupyter

```
!pip install interpolation
```

2.6 Working with Python Files

So far we've focused on executing Python code entered into a Jupyter notebook cell.

Traditionally most Python code has been run in a different way.

Code is first saved in a text file on a local machine

By convention, these text files have a `.py` extension.

We can create an example of such a file as follows:

```
In [2]: %%file foo.py

        print("foobar")
```

```
Overwriting foo.py
```

This writes the line `print("foobar")` into a file called `foo.py` in the local directory.

Here `%%file` is an example of a cell magic.

2.6.1 Editing and Execution

If you come across code saved in a `*.py` file, you'll need to consider the following questions:

1. how should you execute it?
2. How should you modify or edit it?

Option 1: JupyterLab

JupyterLab is an integrated development environment built on top of Jupyter notebooks.

With JupyterLab you can edit and run `*.py` files as well as Jupyter notebooks.

To start JupyterLab, search for it in the applications menu or type `jupyter-lab` in a terminal.

Now you should be able to open, edit and run the file `foo.py` created above by opening it in JupyterLab.

Read the docs or search for a recent YouTube video to find more information.

Option 2: Using a Text Editor

One can also edit files using a text editor and then run them from within Jupyter notebooks.

A text editor is an application that is specifically designed to work with text files — such as Python programs.

Nothing beats the power and efficiency of a good text editor for working with program text.

A good text editor will provide

- efficient text editing commands (e.g., copy, paste, search and replace)
- syntax highlighting, etc.

Right now, an extremely popular text editor for coding is VS Code.

VS Code is easy to use out of the box and has many high quality extensions.

Alternatively, if you want an outstanding free text editor and don't mind a seemingly vertical learning curve plus long days of pain and suffering while all your neural pathways are rewired, try Vim.

2.7 Exercises

2.7.1 Exercise 1

If Jupyter is still running, quit by using `Ctrl-C` at the terminal where you started it.

Now launch again, but this time using `jupyter notebook --no-browser`.

This should start the kernel without launching the browser.

Note also the startup message: It should give you a URL such as `http://localhost:8888` where the notebook is running.

Now

1. Start your browser — or open a new tab if it's already running.
2. Enter the URL from above (e.g. `http://localhost:8888`) in the address bar at the top.

You should now be able to run a standard Jupyter notebook session.

This is an alternative way to start the notebook that can also be handy.

2.7.2 Exercise 2

This exercise will familiarize you with git and GitHub.

Git is a *version control system* — a piece of software used to manage digital projects such as code libraries.

In many cases, the associated collections of files — called *repositories* — are stored on GitHub.

GitHub is a wonderland of collaborative coding projects.

For example, it hosts many of the scientific libraries we'll be using later on, such as this one.

Git is the underlying software used to manage these projects.

Git is an extremely powerful tool for distributed collaboration — for example, we use it to share and synchronize all the source files for these lectures.

There are two main flavors of Git

1. the plain vanilla command line Git version
2. the various point-and-click GUI versions

- See, for example, the GitHub version

As the 1st task, try

1. Installing Git.
2. Getting a copy of QuantEcon.py using Git.

For example, if you've installed the command line version, open up a terminal and enter.

```
git clone https://github.com/QuantEcon/QuantEcon.py.
```

(This is just `git clone` in front of the URL for the repository)

As the 2nd task,

1. Sign up to GitHub.

2. Look into 'forking' GitHub repositories (forking means making your own copy of a GitHub repository, stored on GitHub).

3. Fork QuantEcon.py.

4. Clone your fork to some local directory, make edits, commit them, and push them back up to your forked GitHub repo.

5. If you made a valuable improvement, send us a pull request!

For reading on these and other topics, try

- The official Git documentation.
- Reading through the docs on GitHub.
- Pro Git Book by Scott Chacon and Ben Straub.
- One of the thousands of Git tutorials on the Net.

Chapter 3

An Introductory Example

3.1 Contents

- Overview 3.2
- The Task: Plotting a White Noise Process 3.3
- Version 1 3.4
- Alternative Implementations 3.5
- Another Application 3.6
- Exercises 3.7
- Solutions 3.8

3.2 Overview

We're now ready to start learning the Python language itself.

In this lecture, we will write and then pick apart small Python programs.

The objective is to introduce you to basic Python syntax and data structures.

Deeper concepts will be covered in later lectures.

You should have read the lecture on getting started with Python before beginning this one.

3.3 The Task: Plotting a White Noise Process

Suppose we want to simulate and plot the white noise process $\epsilon_0, \epsilon_1, \ldots, \epsilon_T$, where each draw ϵ_t is independent standard normal.

In other words, we want to generate figures that look something like this:

(Here t is on the horizontal axis and ϵ_t is on the vertical axis.)

We'll do this in several different ways, each time learning something more about Python.

We run the following command first, which helps ensure that plots appear in the notebook if you run it on your own machine.

```
In [1]: %matplotlib inline
```

3.4 Version 1

Here are a few lines of code that perform the task we set

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt

        ϵ_values = np.random.randn(100)
        plt.plot(ϵ_values)
        plt.show()
```

Let's break this program down and see how it works.

3.4.1 Imports

The first two lines of the program import functionality from external code libraries.

The first line imports NumPy, a favorite Python package for tasks like

- working with arrays (vectors and matrices)
- common mathematical functions like `cos` and `sqrt`
- generating random numbers
- linear algebra, etc.

After `import numpy as np` we have access to these attributes via the syntax `np.attribute`.

Here's two more examples

```
In [3]: np.sqrt(4)
```

```
Out[3]: 2.0
```

```
In [4]: np.log(4)
```

```
Out[4]: 1.3862943611198906
```

We could also use the following syntax:

```
In [5]: import numpy

        numpy.sqrt(4)
```

```
Out[5]: 2.0
```

But the former method (using the short name `np`) is convenient and more standard.

Why So Many Imports?

Python programs typically require several import statements.

The reason is that the core language is deliberately kept small, so that it's easy to learn and maintain.

When you want to do something interesting with Python, you almost always need to import additional functionality.

Packages

As stated above, NumPy is a Python *package*.

Packages are used by developers to organize code they wish to share.

In fact, a package is just a directory containing

1. files with Python code — called **modules** in Python speak
2. possibly some compiled code that can be accessed by Python (e.g., functions compiled from C or FORTRAN code)
3. a file called `__init__.py` that specifies what will be executed when we type `import package_name`

In fact, you can find and explore the directory for NumPy on your computer easily enough if you look around.

On this machine, it's located in

```
anaconda3/lib/python3.7/site-packages/numpy
```

Subpackages

Consider the line `ϵ_values = np.random.randn(100)`.

Here `np` refers to the package NumPy, while `random` is a **subpackage** of NumPy.

Subpackages are just packages that are subdirectories of another package.

3.4.2 Importing Names Directly

Recall this code that we saw above

```
In [6]: import numpy as np

        np.sqrt(4)
```

Out[6]: 2.0

Here's another way to access NumPy's square root function

```
In [7]: from numpy import sqrt

        sqrt(4)
```

Out[7]: 2.0

This is also fine.

The advantage is less typing if we use `sqrt` often in our code.

The disadvantage is that, in a long program, these two lines might be separated by many other lines.

Then it's harder for readers to know where `sqrt` came from, should they wish to.

3.4.3 Random Draws

Returning to our program that plots white noise, the remaining three lines after the import statements are

```
In [8]: ϵ_values = np.random.randn(100)
        plt.plot(ϵ_values)
        plt.show()
```

The first line generates 100 (quasi) independent standard normals and stores them in `ϵ_values`.

The next two lines genererate the plot.

We can and will look at various ways to configure and improve this plot below.

3.5 Alternative Implementations

Let's try writing some alternative versions of our first program, which plotted IID draws from the normal distribution.

The programs below are less efficient than the original one, and hence somewhat artificial.

But they do help us illustrate some important Python syntax and semantics in a familiar setting.

3.5.1 A Version with a For Loop

Here's a version that illustrates `for` loops and Python lists.

```
In [9]: ts_length = 100
        ϵ_values = []   # empty list

        for i in range(ts_length):
            e = np.random.randn()
            ϵ_values.append(e)

        plt.plot(ϵ_values)
        plt.show()
```

In brief,

- The first line sets the desired length of the time series.
- The next line creates an empty *list* called `ϵ_values` that will store the ϵ_t values as we generate them.
- The statement `# empty list` is a *comment*, and is ignored by Python's interpreter.
- The next three lines are the `for` loop, which repeatedly draws a new random number ϵ_t and appends it to the end of the list `ϵ_values`.

- The last two lines generate the plot and display it to the user.

Let's study some parts of this program in more detail.

3.5.2 Lists

Consider the statement `ϵ_values = []`, which creates an empty list.

Lists are a *native Python data structure* used to group a collection of objects.

For example, try

```
In [10]: x = [10, 'foo', False]
         type(x)
```

```
Out[10]: list
```

The first element of `x` is an integer, the next is a string, and the third is a Boolean value.

When adding a value to a list, we can use the syntax `list_name.append(some_value)`

```
In [11]: x
```

```
Out[11]: [10, 'foo', False]
```

```
In [12]: x.append(2.5)
         x
```

```
Out[12]: [10, 'foo', False, 2.5]
```

Here `append()` is what's called a *method*, which is a function "attached to" an object—in this case, the list `x`.

We'll learn all about methods later on, but just to give you some idea,

- Python objects such as lists, strings, etc. all have methods that are used to manipulate the data contained in the object.
- String objects have string methods, list objects have list methods, etc.

Another useful list method is `pop()`

```
In [13]: x
```

```
Out[13]: [10, 'foo', False, 2.5]
```

```
In [14]: x.pop()
```

```
Out[14]: 2.5
```

```
In [15]: x
```

```
Out[15]: [10, 'foo', False]
```

Lists in Python are zero-based (as in C, Java or Go), so the first element is referenced by `x[0]`

```
In [16]: x[0]   # first element of x
```

```
Out[16]: 10
```

```
In [17]: x[1]   # second element of x
```

```
Out[17]: 'foo'
```

3.5.3 The For Loop

Now let's consider the `for` loop from the program above, which was

```
In [18]: for i in range(ts_length):
             e = np.random.randn()
             ϵ_values.append(e)
```

Python executes the two indented lines `ts_length` times before moving on.

These two lines are called a `code block`, since they comprise the "block" of code that we are looping over.

Unlike most other languages, Python knows the extent of the code block *only from indentation.*

In our program, indentation decreases after line `ϵ_values.append(e)`, telling Python that this line marks the lower limit of the code block.

More on indentation below—for now, let's look at another example of a `for` loop

```
In [19]: animals = ['dog', 'cat', 'bird']
         for animal in animals:
             print("The plural of " + animal + " is " + animal + "s")
```

```
The plural of dog is dogs
The plural of cat is cats
The plural of bird is birds
```

This example helps to clarify how the `for` loop works: When we execute a loop of the form

```
for variable_name in sequence:
    <code block>
```

The Python interpreter performs the following:

- For each element of the `sequence`, it "binds" the name `variable_name` to that element and then executes the code block.

The `sequence` object can in fact be a very general object, as we'll see soon enough.

3.5.4 A Comment on Indentation

In discussing the `for` loop, we explained that the code blocks being looped over are delimited by indentation.

In fact, in Python, **all** code blocks (i.e., those occurring inside loops, if clauses, function definitions, etc.) are delimited by indentation.

Thus, unlike most other languages, whitespace in Python code affects the output of the program.

Once you get used to it, this is a good thing: It

- forces clean, consistent indentation, improving readability
- removes clutter, such as the brackets or end statements used in other languages

On the other hand, it takes a bit of care to get right, so please remember:

- The line before the start of a code block always ends in a colon
  - `for i in range(10):`
  - `if x > y:`
  - `while x < 100:`
  - etc., etc.
- All lines in a code block **must have the same amount of indentation**.
- The Python standard is 4 spaces, and that's what you should use.

3.5.5 While Loops

The `for` loop is the most common technique for iteration in Python.

But, for the purpose of illustration, let's modify the program above to use a `while` loop instead.

```
In [20]: ts_length = 100
         ϵ_values = []
         i = 0
         while i < ts_length:
             e = np.random.randn()
             ϵ_values.append(e)
             i = i + 1
         plt.plot(ϵ_values)
         plt.show()
```

Note that

- the code block for the `while` loop is again delimited only by indentation
- the statement `i = i + 1` can be replaced by `i += 1`

3.6 Another Application

Let's do one more application before we turn to exercises.

In this application, we plot the balance of a bank account over time.

There are no withdraws over the time period, the last date of which is denoted by T.

The initial balance is b_0 and the interest rate is r.

The balance updates from period t to $t+1$ according to $b_{t+1} = (1+r)b_t$.

In the code below, we generate and plot the sequence $b_0, b_1, \dots, b_T$.

Instead of using a Python list to store this sequence, we will use a NumPy array.

```
In [21]: r = 0.025          # interest rate
         T = 50             # end date
         b = np.empty(T+1)  # an empty NumPy array, to store all b_t
         b[0] = 10          # initial balance

         for t in range(T):
             b[t+1] = (1 + r) * b[t]

         plt.plot(b, label='bank balance')
         plt.legend()
         plt.show()
```

The statement `b = np.empty(T+1)` allocates storage in memory for `T+1` (floating point) numbers.

These numbers are filled in by the `for` loop.

Allocating memory at the start is more efficient than using a Python list and `append`, since the latter must repeatedly ask for storage space from the operating system.

Notice that we added a legend to the plot — a feature you will be asked to use in the exercises.

3.7 Exercises

Now we turn to exercises. It is important that you complete them before continuing, since they present new concepts we will need.

3.7.1 Exercise 1

Your first task is to simulate and plot the correlated time series

$$x_{t+1} = \alpha\, x_t + \epsilon_{t+1} \quad \text{where} \quad x_0 = 0 \quad \text{and} \quad t = 0, \ldots, T$$

The sequence of shocks $\{\epsilon_t\}$ is assumed to be IID and standard normal.

In your solution, restrict your import statements to

```
In [22]: import numpy as np
         import matplotlib.pyplot as plt
```

Set $T = 200$ and $\alpha = 0.9$.

3.7.2 Exercise 2

Starting with your solution to exercise 2, plot three simulated time series, one for each of the cases $\alpha = 0$, $\alpha = 0.8$ and $\alpha = 0.98$.

Use a `for` loop to step through the α values.

If you can, add a legend, to help distinguish between the three time series.

Hints:

- If you call the `plot()` function multiple times before calling `show()`, all of the lines you produce will end up on the same figure.
- For the legend, noted that the expression `'foo' + str(42)` evaluates to `'foo42'`.

3.7.3 Exercise 3

Similar to the previous exercises, plot the time series

$$x_{t+1} = \alpha \left| x_t \right| + \epsilon_{t+1} \quad \text{where} \quad x_0 = 0 \quad \text{and} \quad t = 0, \dots, T$$

Use $T = 200$, $\alpha = 0.9$ and $\{\epsilon_t\}$ as before.

Search online for a function that can be used to compute the absolute value $|x_t|$.

3.7.4 Exercise 4

One important aspect of essentially all programming languages is branching and conditions.

In Python, conditions are usually implemented with if–else syntax.

Here's an example, that prints -1 for each negative number in an array and 1 for each non-negative number

```
In [23]: numbers = [-9, 2.3, -11, 0]
```

```
In [24]: for x in numbers:
             if x < 0:
                 print(-1)
             else:
                 print(1)
```

```
-1
1
-1
1
```

Now, write a new solution to Exercise 3 that does not use an existing function to compute the absolute value.

Replace this existing function with an if–else condition.

3.7.5 Exercise 5

Here's a harder exercise, that takes some thought and planning.

The task is to compute an approximation to π using Monte Carlo.

Use no imports besides

```
In [25]: import numpy as np
```

Your hints are as follows:

- If U is a bivariate uniform random variable on the unit square $(0, 1)^2$, then the probability that U lies in a subset B of $(0, 1)^2$ is equal to the area of B.
- If $U_1, \ldots, U_n$ are IID copies of U, then, as n gets large, the fraction that falls in B, converges to the probability of landing in B.
- For a circle, $area = \pi * radius^2$.

3.8 Solutions

3.8.1 Exercise 1

Here's one solution.

```
In [26]: α = 0.9
         T = 200
         x = np.empty(T+1)
         x[0] = 0

         for t in range(T):
             x[t+1] = α * x[t] + np.random.randn()

         plt.plot(x)
         plt.show()
```

3.8.2 Exercise 2

```
In [27]: α_values = [0.0, 0.8, 0.98]
         T = 200
         x = np.empty(T+1)

         for α in α_values:
             x[0] = 0
             for t in range(T):
                 x[t+1] = α * x[t] + np.random.randn()
             plt.plot(x, label=f'$\\alpha = {α}$')

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

3.8.3 Exercise 3

Here's one solution:

```
In [28]: α = 0.9
         T = 200
         x = np.empty(T+1)
         x[0] = 0

         for t in range(T):
             x[t+1] = α * np.abs(x[t]) + np.random.randn()

         plt.plot(x)
         plt.show()
```

3.8.4 Exercise 4

Here's one way:

```
In [29]: α = 0.9
         T = 200
         x = np.empty(T+1)
         x[0] = 0

         for t in range(T):
             if x[t] < 0:
                 abs_x = - x[t]
             else:
                 abs_x = x[t]
             x[t+1] = α * abs_x + np.random.randn()

         plt.plot(x)
         plt.show()
```

Here's a shorter way to write the same thing:

```
In [30]: α = 0.9
         T = 200
         x = np.empty(T+1)
         x[0] = 0

         for t in range(T):
             abs_x = - x[t] if x[t] < 0 else x[t]
             x[t+1] = α * abs_x + np.random.randn()

         plt.plot(x)
         plt.show()
```

3.8.5 Exercise 5

Consider the circle of diameter 1 embedded in the unit square.

Let A be its area and let $r = 1/2$ be its radius.

If we know π then we can compute A via $A = \pi r^2$.

But here the point is to compute π, which we can do by $\pi = A/r^2$.

Summary: If we can estimate the area of a circle with diameter 1, then dividing by $r^2 = (1/2)^2 = 1/4$ gives an estimate of π.

We estimate the area by sampling bivariate uniforms and looking at the fraction that falls into the circle.

In [31]:
```
n = 100000

count = 0
for i in range(n):
    u, v = np.random.uniform(), np.random.uniform()
    d = np.sqrt((u - 0.5)**2 + (v - 0.5)**2)
    if d < 0.5:
        count += 1

area_estimate = count / n

print(area_estimate * 4)  # dividing by radius**2
```

```
3.1412
```

Chapter 4

Functions

4.1 Contents

- Overview 4.2
- Function Basics 4.3
- Defining Functions 4.4
- Applications 4.5
- Exercises 4.6
- Solutions 4.7

4.2 Overview

One construct that's extremely useful and provided by almost all programming languages is **functions**.

We have already met several functions, such as

- the `sqrt()` function from NumPy and
- the built-in `print()` function

In this lecture we'll treat functions systematically and begin to learn just how useful and important they are.

One of the things we will learn to do is build our own user-defined functions

We will use the following imports.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

4.3 Function Basics

A function is a named section of a program that implements a specific task.

Many functions exist already and we can use them off the shelf.

First we review these functions and then discuss how we can build our own.

4.3.1 Built-In Functions

Python has a number of *built-in* functions that are available without `import`.

We have already met some

```
In [2]: max(19, 20)
```

```
Out[2]: 20
```

```
In [3]: print('foobar')
```

```
foobar
```

```
In [4]: str(22)
```

```
Out[4]: '22'
```

```
In [5]: type(22)
```

```
Out[5]: int
```

Two more useful built-in functions are `any()` and `all()`

```
In [6]: bools = False, True, True
        all(bools)  # True if all are True and False otherwise
```

```
Out[6]: False
```

```
In [7]: any(bools)  # False if all are False and True otherwise
```

```
Out[7]: True
```

The full list of Python built-ins is here.

4.3.2 Third Party Functions

If the built-in functions don't cover what we need, we either need to import functions or create our own.

Examples of importing and using functions were given in the previous lecture

Here's another one, which tests whether a given year is a leap year:

```
In [8]: import calendar

        calendar.isleap(2020)
```

```
Out[8]: True
```

4.4 Defining Functions

In many instances, it is useful to be able to define our own functions.

This will become clearer as you see more examples.

Let's start by discussing how it's done.

4.4.1 Syntax

Here's a very simple Python function, that implements the mathematical function $f(x) = 2x + 1$

```
In [9]: def f(x):
            return 2 * x + 1
```

Now that we've *defined* this function, let's *call* it and check whether it does what we expect:

```
In [10]: f(1)

Out[10]: 3

In [11]: f(10)

Out[11]: 21
```

Here's a longer function, that computes the absolute value of a given number.

(Such a function already exists as a built-in, but let's write our own for the exercise.)

```
In [12]: def new_abs_function(x):

             if x < 0:
                 abs_value = -x
             else:
                 abs_value = x

             return abs_value
```

Let's review the syntax here.

- `def` is a Python keyword used to start function definitions.
- `def new_abs_function(x):` indicates that the function is called `new_abs_function` and that it has a single argument `x`.
- The indented code is a code block called the *function body*.
- The `return` keyword indicates that `abs_value` is the object that should be returned to the calling code.

This whole function definition is read by the Python interpreter and stored in memory.

Let's call it to check that it works:

```
In [13]: print(new_abs_function(3))
         print(new_abs_function(-3))
```

```
    3
3
```

4.4.2 Why Write Functions?

User-defined functions are important for improving the clarity of your code by

- separating different strands of logic
- facilitating code reuse

(Writing the same thing twice is almost always a bad idea)

We will say more about this later.

4.5 Applications

4.5.1 Random Draws

Consider again this code from the previous lecture

```
In [14]: ts_length = 100
         ϵ_values = []   # empty list

         for i in range(ts_length):
             e = np.random.randn()
             ϵ_values.append(e)

         plt.plot(ϵ_values)
         plt.show()
```

We will break this program into two parts:

1. A user-defined function that generates a list of random variables.
2. The main part of the program that
3. calls this function to get data
4. plots the data

This is accomplished in the next program

```
In [15]: def generate_data(n):
             ϵ_values = []
             for i in range(n):
                 e = np.random.randn()
                 ϵ_values.append(e)
             return ϵ_values

         data = generate_data(100)
         plt.plot(data)
         plt.show()
```

When the interpreter gets to the expression `generate_data(100)`, it executes the function body with `n` set equal to 100.

The net result is that the name `data` is *bound* to the list `ϵ_values` returned by the function.

4.5.2 Adding Conditions

Our function `generate_data()` is rather limited.

Let's make it slightly more useful by giving it the ability to return either standard normals or uniform random variables on $(0, 1)$ as required.

This is achieved in the next piece of code.

```
In [16]: def generate_data(n, generator_type):
             ϵ_values = []
             for i in range(n):
                 if generator_type == 'U':
                     e = np.random.uniform(0, 1)
                 else:
                     e = np.random.randn()
                 ϵ_values.append(e)
             return ϵ_values

         data = generate_data(100, 'U')
         plt.plot(data)
         plt.show()
```

Hopefully, the syntax of the if/else clause is self-explanatory, with indentation again delimiting the extent of the code blocks.

Notes

- We are passing the argument `U` as a string, which is why we write it as `'U'`.
- Notice that equality is tested with the `==` syntax, not `=`.
  - For example, the statement `a = 10` assigns the name `a` to the value `10`.
  - The expression `a == 10` evaluates to either `True` or `False`, depending on the value of `a`.

Now, there are several ways that we can simplify the code above.

For example, we can get rid of the conditionals all together by just passing the desired generator type *as a function*.

To understand this, consider the following version.

```
In [17]: def generate_data(n, generator_type):
             ϵ_values = []
             for i in range(n):
                 e = generator_type()
                 ϵ_values.append(e)
             return ϵ_values

         data = generate_data(100, np.random.uniform)
         plt.plot(data)
         plt.show()
```

Now, when we call the function `generate_data()`, we pass `np.random.uniform` as the second argument.

This object is a *function*.

When the function call `generate_data(100, np.random.uniform)` is executed, Python runs the function code block with `n` equal to 100 and the name `generator_type` "bound" to the function `np.random.uniform`.

- While these lines are executed, the names `generator_type` and `np.random.uniform` are "synonyms", and can be used in identical ways.

This principle works more generally—for example, consider the following piece of code

```
In [18]: max(7, 2, 4)   # max() is a built-in Python function

Out[18]: 7

In [19]: m = max
         m(7, 2, 4)

Out[19]: 7
```

Here we created another name for the built-in function `max()`, which could then be used in identical ways.

In the context of our program, the ability to bind new names to functions means that there is no problem *passing a function as an argument to another function*—as we did above.

4.6 Exercises

4.6.1 Exercise 1

Recall that $n!$ is read as "n factorial" and defined as $n! = n \times (n-1) \times \cdots \times 2 \times 1$.

There are functions to compute this in various modules, but let's write our own version as an exercise.

In particular, write a function `factorial` such that `factorial(n)` returns $n!$ for any positive integer n.

4.6.2 Exercise 2

The binomial random variable $Y \sim Bin(n, p)$ represents the number of successes in n binary trials, where each trial succeeds with probability p.

Without any import besides `from numpy.random import uniform`, write a function `binomial_rv` such that `binomial_rv(n, p)` generates one draw of Y.

Hint: If U is uniform on $(0, 1)$ and $p \in (0, 1)$, then the expression `U < p` evaluates to `True` with probability p.

4.6.3 Exercise 3

First, write a function that returns one realization of the following random device

1. Flip an unbiased coin 10 times.
2. If a head occurs `k` or more times consecutively within this sequence at least once, pay one dollar.
3. If not, pay nothing.

Second, write another function that does the same task except that the second rule of the above random device becomes

- If a head occurs `k` or more times within this sequence, pay one dollar.

Use no import besides `from numpy.random import uniform`.

4.7 Solutions

4.7.1 Exercise 1

Here's one solution.

In [20]:
```
def factorial(n):
    k = 1
    for i in range(n):
        k = k * (i + 1)
    return k

factorial(4)
```

Out[20]: 24

4.7.2 Exercise 2

In [21]:
```
from numpy.random import uniform

def binomial_rv(n, p):
    count = 0
    for i in range(n):
        U = uniform()
        if U < p:
            count = count + 1    # Or count += 1
    return count

binomial_rv(10, 0.5)
```

Out[21]: 8

4.7.3 Exercise 3

Here's a function for the first random device.

In [22]:
```
from numpy.random import uniform

def draw(k):  # pays if k consecutive successes in a sequence

    payoff = 0
    count = 0

    for i in range(10):
        U = uniform()
        count = count + 1 if U < 0.5 else 0
        print(count)    # print counts for clarity
        if count == k:
            payoff = 1

    return payoff

draw(3)
```

```
0
0
1
2
0
1
```

```
2
0
0
1
```

Out[22]: 0

Here's another function for the second random device.

In [23]:
```
def draw_new(k):  # pays if k successes in a sequence

    payoff = 0
    count = 0

    for i in range(10):
        U = uniform()
        count = count + ( 1 if U < 0.5 else 0 )
        print(count)
        if count == k:
            payoff = 1

    return payoff

draw_new(3)
```

```
  1
1
2
3
4
4
4
5
6
7
```

Out[23]: 1

Chapter 5

Python Essentials

5.1 Contents

- Overview 5.2
- Data Types 5.3
- Input and Output 5.4
- Iterating 5.5
- Comparisons and Logical Operators 5.6
- More Functions 5.7
- Coding Style and PEP8 5.8
- Exercises 5.9
- Solutions 5.10

5.2 Overview

We have covered a lot of material quite quickly, with a focus on examples.

Now let's cover some core features of Python in a more systematic way.

This approach is less exciting but helps clear up some details.

5.3 Data Types

Computer programs typically keep track of a range of data types.

For example, `1.5` is a floating point number, while `1` is an integer.

Programs need to distinguish between these two types for various reasons.

One is that they are stored in memory differently.

Another is that arithmetic operations are different

- For example, floating point arithmetic is implemented on most machines by a specialized Floating Point Unit (FPU).

In general, floats are more informative but arithmetic operations on integers are faster and more accurate.

Python provides numerous other built-in Python data types, some of which we've already met

- strings, lists, etc.

Let's learn a bit more about them.

5.3.1 Primitive Data Types

One simple data type is **Boolean values**, which can be either `True` or `False`

```
In [1]: x = True
        x
```

```
Out[1]: True
```

We can check the type of any object in memory using the `type()` function.

```
In [2]: type(x)
```

```
Out[2]: bool
```

In the next line of code, the interpreter evaluates the expression on the right of = and binds y to this value

```
In [3]: y = 100 < 10
        y
```

```
Out[3]: False
```

```
In [4]: type(y)
```

```
Out[4]: bool
```

In arithmetic expressions, `True` is converted to `1` and `False` is converted `0`.

This is called **Boolean arithmetic** and is often useful in programming.

Here are some examples

```
In [5]: x + y
```

```
Out[5]: 1
```

```
In [6]: x * y
```

```
Out[6]: 0
```

```
In [7]: True + True
```

```
Out[7]: 2
```

```
In [8]: bools = [True, True, False, True]  # List of Boolean values

        sum(bools)
```

```
Out[8]: 3
```

Complex numbers are another primitive data type in Python

```
In [9]: x = complex(1, 2)
        y = complex(2, 1)
        print(x * y)

        type(x)
```

```
5j
```

```
Out[9]: complex
```

5.3.2 Containers

Python has several basic types for storing collections of (possibly heterogeneous) data.

We've already discussed lists.

A related data type is **tuples**, which are "immutable" lists

```
In [10]: x = ('a', 'b')  # Parentheses instead of the square brackets
         x = 'a', 'b'    # Or no brackets --- the meaning is identical
         x
```

```
Out[10]: ('a', 'b')
```

```
In [11]: type(x)
```

```
Out[11]: tuple
```

In Python, an object is called **immutable** if, once created, the object cannot be changed.

Conversely, an object is **mutable** if it can still be altered after creation.

Python lists are mutable

```
In [12]: x = [1, 2]
         x[0] = 10
         x
```

```
Out[12]: [10, 2]
```

But tuples are not

```
In [13]: x = (1, 2)
         x[0] = 10
```

```
         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-13-d1b2647f6c81> in <module>
           1 x = (1, 2)
       ----> 2 x[0] = 10


         TypeError: 'tuple' object does not support item assignment
```

We'll say more about the role of mutable and immutable data a bit later.

Tuples (and lists) can be "unpacked" as follows

```
In [ ]: integers = (10, 20, 30)
        x, y, z = integers
        x
```

```
In [14]: y
```

```
Out[14]: (2+1j)
```

You've actually seen an example of this already.

Tuple unpacking is convenient and we'll use it often.

Slice Notation

To access multiple elements of a list or tuple, you can use Python's slice notation.

For example,

```
In [15]: a = [2, 4, 6, 8]
         a[1:]
```

```
Out[15]: [4, 6, 8]
```

```
In [16]: a[1:3]
```

```
Out[16]: [4, 6]
```

The general rule is that `a[m:n]` returns `n - m` elements, starting at `a[m]`.

Negative numbers are also permissible

```
In [17]: a[-2:]  # Last two elements of the list
```

```
Out[17]: [6, 8]
```

The same slice notation works on tuples and strings

```
In [18]: s = 'foobar'
         s[-3:]  # Select the last three elements
```

```
Out[18]: 'bar'
```

Sets and Dictionaries

Two other container types we should mention before moving on are sets and dictionaries.

Dictionaries are much like lists, except that the items are named instead of numbered

```
In [19]: d = {'name': 'Frodo', 'age': 33}
         type(d)
```

```
Out[19]: dict
```

```
In [20]: d['age']
```

```
Out[20]: 33
```

The names `'name'` and `'age'` are called the *keys*.

The objects that the keys are mapped to (`'Frodo'` and `33`) are called the `values`.

Sets are unordered collections without duplicates, and set methods provide the usual set-theoretic operations

```
In [21]: s1 = {'a', 'b'}
         type(s1)
```

```
Out[21]: set
```

```
In [22]: s2 = {'b', 'c'}
         s1.issubset(s2)
```

```
Out[22]: False
```

```
In [23]: s1.intersection(s2)
```

```
Out[23]: {'b'}
```

The `set()` function creates sets from sequences

```
In [24]: s3 = set(('foo', 'bar', 'foo'))
         s3
```

```
Out[24]: {'bar', 'foo'}
```

5.4 Input and Output

Let's briefly review reading and writing to text files, starting with writing

```
In [25]: f = open('newfile.txt', 'w')   # Open 'newfile.txt' for writing
         f.write('Testing\n')           # Here '\n' means new line
         f.write('Testing again')
         f.close()
```

Here

- The built-in function `open()` creates a file object for writing to.
- Both `write()` and `close()` are methods of file objects.

Where is this file that we've created?

Recall that Python maintains a concept of the present working directory (pwd) that can be located from with Jupyter or IPython via

```
In [26]: %pwd
```

```
Out[26]: '/home/ubuntu/repos/lecture-source-py/_build/jupyterpdf/executed'
```

If a path is not specified, then this is where Python writes to.

We can also use Python to read the contents of `newline.txt` as follows

```
In [27]: f = open('newfile.txt', 'r')
         out = f.read()
         out
```

```
Out[27]: 'Testing\nTesting again'
```

```
In [28]: print(out)
```

```
Testing
Testing again
```

5.4.1 Paths

Note that if `newfile.txt` is not in the present working directory then this call to `open()` fails.

In this case, you can shift the file to the pwd or specify the full path to the file

```
f = open('insert_full_path_to_file/newfile.txt', 'r')
```

5.5 Iterating

One of the most important tasks in computing is stepping through a sequence of data and performing a given action.

One of Python's strengths is its simple, flexible interface to this kind of iteration via the `for` loop.

5.5.1 Looping over Different Objects

Many Python objects are "iterable", in the sense that they can be looped over.

To give an example, let's write the file us_cities.txt, which lists US cities and their population, to the present working directory.

```
In [29]: %%file us_cities.txt
         new york: 8244910
         los angeles: 3819702
         chicago: 2707120
         houston: 2145146
         philadelphia: 1536471
         phoenix: 1469471
         san antonio: 1359758
         san diego: 1326179
         dallas: 1223229
```

```
Overwriting us_cities.txt
```

Here %%file is an IPython cell magic.

Suppose that we want to make the information more readable, by capitalizing names and adding commas to mark thousands.

The program below reads the data in and makes the conversion:

```
In [30]: data_file = open('us_cities.txt', 'r')
         for line in data_file:
             city, population = line.split(':')         # Tuple unpacking
             city = city.title()                        # Capitalize city names
             population = f'{int(population):,}'        # Add commas to numbers
             print(city.ljust(15) + population)
         data_file.close()
```

```
New York       8,244,910
Los Angeles    3,819,702
Chicago        2,707,120
Houston        2,145,146
Philadelphia   1,536,471
Phoenix        1,469,471
San Antonio    1,359,758
San Diego      1,326,179
Dallas         1,223,229
```

Here `format()` is a string method used for inserting variables into strings.

The reformatting of each line is the result of three different string methods, the details of which can be left till later.

The interesting part of this program for us is line 2, which shows that

1. The file object `data_file` is iterable, in the sense that it can be placed to the right of `in` within a `for` loop.

2. Iteration steps through each line in the file.

This leads to the clean, convenient syntax shown in our program.

Many other kinds of objects are iterable, and we'll discuss some of them later on.

5.5.2 Looping without Indices

One thing you might have noticed is that Python tends to favor looping without explicit indexing.

For example,

```
In [31]: x_values = [1, 2, 3]  # Some iterable x
         for x in x_values:
             print(x * x)
```

```
1
4
9
```

is preferred to

```
In [32]: for i in range(len(x_values)):
             print(x_values[i] * x_values[i])
```

```
1
4
9
```

When you compare these two alternatives, you can see why the first one is preferred.

Python provides some facilities to simplify looping without indices.

One is `zip()`, which is used for stepping through pairs from two sequences.

For example, try running the following code

```
In [33]: countries = ('Japan', 'Korea', 'China')
         cities = ('Tokyo', 'Seoul', 'Beijing')
         for country, city in zip(countries, cities):
             print(f'The capital of {country} is {city}')
```

```
The capital of Japan is Tokyo
The capital of Korea is Seoul
The capital of China is Beijing
```

The `zip()` function is also useful for creating dictionaries — for example

```
In [34]: names = ['Tom', 'John']
         marks = ['E', 'F']
         dict(zip(names, marks))
```

```
Out[34]: {'Tom': 'E', 'John': 'F'}
```

If we actually need the index from a list, one option is to use `enumerate()`.

To understand what `enumerate()` does, consider the following example

```
In [35]: letter_list = ['a', 'b', 'c']
         for index, letter in enumerate(letter_list):
             print(f"letter_list[{index}] = '{letter}'")
```

```
letter_list[0] = 'a'
letter_list[1] = 'b'
letter_list[2] = 'c'
```

5.5.3 List Comprehensions

We can also simplify the code for generating the list of random draws considerably by using something called a *list comprehension*.

List comprehensions are an elegant Python tool for creating lists.

Consider the following example, where the list comprehension is on the right-hand side of the second line

```
In [36]: animals = ['dog', 'cat', 'bird']
         plurals = [animal + 's' for animal in animals]
         plurals
```

```
Out[36]: ['dogs', 'cats', 'birds']
```

Here's another example

```
In [37]: range(8)
```

```
Out[37]: range(0, 8)
```

```
In [38]: doubles = [2 * x for x in range(8)]
         doubles
```

```
Out[38]: [0, 2, 4, 6, 8, 10, 12, 14]
```

5.6 Comparisons and Logical Operators

5.6.1 Comparisons

Many different kinds of expressions evaluate to one of the Boolean values (i.e., `True` or `False`).

A common type is comparisons, such as

```
In [39]: x, y = 1, 2
         x < y

Out[39]: True

In [40]: x > y

Out[40]: False
```

One of the nice features of Python is that we can *chain* inequalities

```
In [41]: 1 < 2 < 3

Out[41]: True

In [42]: 1 <= 2 <= 3

Out[42]: True
```

As we saw earlier, when testing for equality we use `==`

```
In [43]: x = 1    # Assignment
         x == 2   # Comparison

Out[43]: False
```

For "not equal" use `!=`

```
In [44]: 1 != 2

Out[44]: True
```

Note that when testing conditions, we can use **any** valid Python expression

```
In [45]: x = 'yes' if 42 else 'no'
         x

Out[45]: 'yes'

In [46]: x = 'yes' if [] else 'no'
         x

Out[46]: 'no'
```

What's going on here?

The rule is:

- Expressions that evaluate to zero, empty sequences or containers (strings, lists, etc.) and `None` are all equivalent to `False`.
  - for example, `[]` and `()` are equivalent to `False` in an `if` clause
- All other values are equivalent to `True`.
  - for example, `42` is equivalent to `True` in an `if` clause

5.6.2 Combining Expressions

We can combine expressions using `and`, `or` and `not`.

These are the standard logical connectives (conjunction, disjunction and denial)

```
In [47]: 1 < 2 and 'f' in 'foo'

Out[47]: True

In [48]: 1 < 2 and 'g' in 'foo'

Out[48]: False

In [49]: 1 < 2 or 'g' in 'foo'

Out[49]: True

In [50]: not True

Out[50]: False

In [51]: not not True

Out[51]: True
```

Remember

- `P and Q` is `True` if both are `True`, else `False`
- `P or Q` is `False` if both are `False`, else `True`

5.7 More Functions

Let's talk a bit more about functions, which are all important for good programming style.

5.7.1 The Flexibility of Python Functions

As we discussed in the previous lecture, Python functions are very flexible.

In particular

- Any number of functions can be defined in a given file.
- Functions can be (and often are) defined inside other functions.
- Any object can be passed to a function as an argument, including other functions.
- A function can return any kind of object, including functions.

We already gave an example of how straightforward it is to pass a function to a function.

Note that a function can have arbitrarily many `return` statements (including zero).

Execution of the function terminates when the first return is hit, allowing code like the following example

```
In [52]: def f(x):
             if x < 0:
                 return 'negative'
             return 'nonnegative'
```

Functions without a return statement automatically return the special Python object `None`.

5.7.2 Docstrings

Python has a system for adding comments to functions, modules, etc. called *docstrings*.

The nice thing about docstrings is that they are available at run-time.

Try running this

```
In [53]: def f(x):
             """
             This function squares its argument
             """
             return x**2
```

After running this code, the docstring is available

```
In [54]: f?
```

```
Type:       function
String Form:<function f at 0x2223320>
File:       /home/john/temp/temp.py
Definition: f(x)
Docstring:  This function squares its argument
```

```
In [55]: f??
```

```
Type:       function
String Form:<function f at 0x2223320>
File:       /home/john/temp/temp.py
Definition: f(x)
Source:
def f(x):
    """
    This function squares its argument
    """
    return x**2
```

With one question mark we bring up the docstring, and with two we get the source code as well.

5.7.3 One-Line Functions: `lambda`

The `lambda` keyword is used to create simple functions on one line.

For example, the definitions

```
In [56]: def f(x):
             return x**3
```

and

```
In [57]: f = lambda x: x**3
```

are entirely equivalent.

To see why `lambda` is useful, suppose that we want to calculate $\int_0^2 x^3 dx$ (and have forgotten our high-school calculus).

The SciPy library has a function called `quad` that will do this calculation for us.

The syntax of the `quad` function is `quad(f, a, b)` where `f` is a function and `a` and `b` are numbers.

To create the function $f(x) = x^3$ we can use `lambda` as follows

```
In [58]: from scipy.integrate import quad

         quad(lambda x: x**3, 0, 2)
```

```
Out[58]: (4.0, 4.440892098500626e-14)
```

Here the function created by `lambda` is said to be *anonymous* because it was never given a name.

5.7.4 Keyword Arguments

In a previous lecture, you came across the statement

```
plt.plot(x, 'b-', label="white noise")
```

In this call to Matplotlib's `plot` function, notice that the last argument is passed in `name=argument` syntax.

This is called a *keyword argument*, with `label` being the keyword.

Non-keyword arguments are called *positional arguments*, since their meaning is determined by order

- `plot(x, 'b-', label="white noise")` is different from `plot('b-', x, label="white noise")`

Keyword arguments are particularly useful when a function has a lot of arguments, in which case it's hard to remember the right order.

You can adopt keyword arguments in user-defined functions with no difficulty.

The next example illustrates the syntax

```
In [59]: def f(x, a=1, b=1):
             return a + b * x
```

The keyword argument values we supplied in the definition of `f` become the default values

```
In [60]: f(2)
```

```
Out[60]: 3
```

They can be modified as follows

```
In [61]: f(2, a=4, b=5)
```

```
Out[61]: 14
```

5.8 Coding Style and PEP8

To learn more about the Python programming philosophy type `import this` at the prompt.

Among other things, Python strongly favors consistency in programming style.

We've all heard the saying about consistency and little minds.

In programming, as in mathematics, the opposite is true

- A mathematical paper where the symbols $\cup$ and $\cap$ were reversed would be very hard to read, even if the author told you so on the first page.

In Python, the standard style is set out in PEP8.

(Occasionally we'll deviate from PEP8 in these lectures to better match mathematical notation)

5.9 Exercises

Solve the following exercises.

(For some, the built-in function `sum()` comes in handy).

5.9.1 Exercise 1

Part 1: Given two numeric lists or tuples `x_vals` and `y_vals` of equal length, compute their inner product using `zip()`.

Part 2: In one line, count the number of even numbers in 0,…,99.

- Hint: `x % 2` returns 0 if `x` is even, 1 otherwise.

Part 3: Given `pairs = ((2, 5), (4, 2), (9, 8), (12, 10))`, count the number of pairs `(a, b)` such that both `a` and `b` are even.

5.9.2 Exercise 2

Consider the polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n = \sum_{i=0}^{n} a_i x^i \tag{1}$$

Write a function `p` such that `p(x, coeff)` that computes the value in (1) given a point `x` and a list of coefficients `coeff`.

Try to use `enumerate()` in your loop.

5.9.3 Exercise 3

Write a function that takes a string as an argument and returns the number of capital letters in the string.

Hint: `'foo'.upper()` returns `'FOO'`.

5.9.4 Exercise 4

Write a function that takes two sequences `seq_a` and `seq_b` as arguments and returns `True` if every element in `seq_a` is also an element of `seq_b`, else `False`.

- By "sequence" we mean a list, a tuple or a string.
- Do the exercise without using sets and set methods.

5.9.5 Exercise 5

When we cover the numerical libraries, we will see they include many alternatives for interpolation and function approximation.

Nevertheless, let's write our own function approximation routine as an exercise.

In particular, without using any imports, write a function `linapprox` that takes as arguments

- A function `f` mapping some interval $[a, b]$ into $\mathbb{R}$.
- Two scalars `a` and `b` providing the limits of this interval.
- An integer `n` determining the number of grid points.
- A number `x` satisfying `a <= x <= b`.

and returns the piecewise linear interpolation of `f` at `x`, based on `n` evenly spaced grid points `a = point[0] < point[1] < ... < point[n-1] = b`.

Aim for clarity, not efficiency.

5.9.6 Exercise 6

Using list comprehension syntax, we can simplify the loop in the following code.

```
In [62]: import numpy as np

         n = 100
         ϵ_values = []
         for i in range(n):
```

```
    e = np.random.randn()
    ϵ_values.append(e)
```

5.10 Solutions

5.10.1 Exercise 1

Part 1 Solution:

Here's one possible solution

```
In [63]: x_vals = [1, 2, 3]
         y_vals = [1, 1, 1]
         sum([x * y for x, y in zip(x_vals, y_vals)])

Out[63]: 6
```

This also works

```
In [64]: sum(x * y for x, y in zip(x_vals, y_vals))

Out[64]: 6
```

Part 2 Solution:

One solution is

```
In [65]: sum([x % 2 == 0 for x in range(100)])

Out[65]: 50
```

This also works:

```
In [66]: sum(x % 2 == 0 for x in range(100))

Out[66]: 50
```

Some less natural alternatives that nonetheless help to illustrate the flexibility of list comprehensions are

```
In [67]: len([x for x in range(100) if x % 2 == 0])

Out[67]: 50
```

and

```
In [68]: sum([1 for x in range(100) if x % 2 == 0])

Out[68]: 50
```

Part 3 Solution

Here's one possibility

```
In [69]: pairs = ((2, 5), (4, 2), (9, 8), (12, 10))
         sum([x % 2 == 0 and y % 2 == 0 for x, y in pairs])
```

```
Out[69]: 2
```

5.10.2 Exercise 2

```
In [70]: def p(x, coeff):
             return sum(a * x**i for i, a in enumerate(coeff))
```

```
In [71]: p(1, (2, 4))
```

```
Out[71]: 6
```

5.10.3 Exercise 3

Here's one solution:

```
In [72]: def f(string):
             count = 0
             for letter in string:
                 if letter == letter.upper() and letter.isalpha():
                     count += 1
             return count

         f('The Rain in Spain')
```

```
Out[72]: 3
```

An alternative, more pythonic solution:

```
In [73]: def count_uppercase_chars(s):
             return sum([c.isupper() for c in s])

         count_uppercase_chars('The Rain in Spain')
```

```
Out[73]: 3
```

5.10.4 Exercise 4

Here's a solution:

```
In [74]: def f(seq_a, seq_b):
             is_subset = True
             for a in seq_a:
```

```
        if a not in seq_b:
            is_subset = False
    return is_subset

# == test == #

print(f([1, 2], [1, 2, 3]))
print(f([1, 2, 3], [1, 2]))
```

```
True
False
```

Of course, if we use the `sets` data type then the solution is easier

```
In [75]: def f(seq_a, seq_b):
             return set(seq_a).issubset(set(seq_b))
```

5.10.5 Exercise 5

```
In [76]: def linapprox(f, a, b, n, x):
             """
             Evaluates the piecewise linear interpolant of f at x on the interval
             [a, b], with n evenly spaced grid points.

             Parameters
             ==========
                 f : function
                     The function to approximate

                 x, a, b : scalars (floats or integers)
                     Evaluation point and endpoints, with a <= x <= b

                 n : integer
                     Number of grid points

             Returns
             =======
                 A float. The interpolant evaluated at x

             """
             length_of_interval = b - a
             num_subintervals = n - 1
             step = length_of_interval / num_subintervals

             # === find first grid point larger than x === #
             point = a
             while point <= x:
                 point += step

             # === x must lie between the gridpoints (point - step) and point === #
             u, v = point - step, point

             return f(u) + (x - u) * (f(v) - f(u)) / (v - u)
```

5.10.6 Exercise 6

Here's one solution.

```
In [77]: n = 100
         ϵ_values = [np.random.randn() for i in range(n)]
```

Chapter 6

OOP I: Introduction to Object Oriented Programming

6.1 Contents

- Overview 6.2
- Objects 6.3
- Summary 6.4

6.2 Overview

OOP is one of the major paradigms in programming.

The traditional programming paradigm (think Fortran, C, MATLAB, etc.) is called *procedural*.

It works as follows

- The program has a state corresponding to the values of its variables.
- Functions are called to act on these data.
- Data are passed back and forth via function calls.

In contrast, in the OOP paradigm

- data and functions are "bundled together" into "objects"

(Functions in this context are referred to as **methods**)

6.2.1 Python and OOP

Python is a pragmatic language that blends object-oriented and procedural styles, rather than taking a purist approach.

However, at a foundational level, Python *is* object-oriented.

In particular, in Python, *everything is an object*.

In this lecture, we explain what that statement means and why it matters.

6.3 Objects

In Python, an *object* is a collection of data and instructions held in computer memory that consists of

1. a type
2. a unique identity
3. data (i.e., content)
4. methods

These concepts are defined and discussed sequentially below.

6.3.1 Type

Python provides for different types of objects, to accommodate different categories of data.

For example

```
In [1]: s = 'This is a string'
        type(s)
```

```
Out[1]: str
```

```
In [2]: x = 42   # Now let's create an integer
        type(x)
```

```
Out[2]: int
```

The type of an object matters for many expressions.

For example, the addition operator between two strings means concatenation

```
In [3]: '300' + 'cc'
```

```
Out[3]: '300cc'
```

On the other hand, between two numbers it means ordinary addition

```
In [4]: 300 + 400
```

```
Out[4]: 700
```

Consider the following expression

```
In [5]: '300' + 400
```

```
↪---------------------------------------------------------------------------

    TypeError                                 Traceback (most
↪recent call last)

    <ipython-input-5-263a89d2d982> in <module>
----> 1 '300' + 400


    TypeError: can only concatenate str (not "int") to str
```

Here we are mixing types, and it's unclear to Python whether the user wants to

- convert `'300'` to an integer and then add it to `400`, or
- convert `400` to string and then concatenate it with `'300'`

Some languages might try to guess but Python is *strongly typed*

- Type is important, and implicit type conversion is rare.
- Python will respond instead by raising a `TypeError`.

To avoid the error, you need to clarify by changing the relevant type.

For example,

```
In [ ]: int('300') + 400   # To add as numbers, change the string to an integer
```

6.3.2 Identity

In Python, each object has a unique identifier, which helps Python (and us) keep track of the object.

The identity of an object can be obtained via the `id()` function

```
In [6]: y = 2.5
        z = 2.5
        id(y)

Out[6]: 139721126716712

In [7]: id(z)

Out[7]: 139721126716760
```

In this example, `y` and `z` happen to have the same value (i.e., `2.5`), but they are not the same object.

The identity of an object is in fact just the address of the object in memory.

6.3.3 Object Content: Data and Attributes

If we set `x = 42` then we create an object of type `int` that contains the data `42`.

In fact, it contains more, as the following example shows

```
In [8]: x = 42
        x

Out[8]: 42

In [9]: x.imag

Out[9]: 0

In [10]: x.__class__

Out[10]: int
```

When Python creates this integer object, it stores with it various auxiliary information, such as the imaginary part, and the type.

Any name following a dot is called an *attribute* of the object to the left of the dot.

- e.g.,`imag` and `__class__` are attributes of `x`.

We see from this example that objects have attributes that contain auxiliary information.

They also have attributes that act like functions, called *methods*.

These attributes are important, so let's discuss them in-depth.

6.3.4 Methods

Methods are *functions that are bundled with objects*.

Formally, methods are attributes of objects that are callable (i.e., can be called as functions)

```
In [11]: x = ['foo', 'bar']
         callable(x.append)

Out[11]: True

In [12]: callable(x.__doc__)

Out[12]: False
```

Methods typically act on the data contained in the object they belong to, or combine that data with other data

```
In [13]: x = ['a', 'b']
         x.append('c')
         s = 'This is a string'
         s.upper()
```

```
Out[13]: 'THIS IS A STRING'
```

```
In [14]: s.lower()
```

```
Out[14]: 'this is a string'
```

```
In [15]: s.replace('This', 'That')
```

```
Out[15]: 'That is a string'
```

A great deal of Python functionality is organized around method calls.

For example, consider the following piece of code

```
In [16]: x = ['a', 'b']
         x[0] = 'aa'  # Item assignment using square bracket notation
         x
```

```
Out[16]: ['aa', 'b']
```

It doesn't look like there are any methods used here, but in fact the square bracket assignment notation is just a convenient interface to a method call.

What actually happens is that Python calls the `__setitem__` method, as follows

```
In [17]: x = ['a', 'b']
         x.__setitem__(0, 'aa')  # Equivalent to x[0] = 'aa'
         x
```

```
Out[17]: ['aa', 'b']
```

(If you wanted to you could modify the `__setitem__` method, so that square bracket assignment does something totally different)

6.4 Summary

In Python, *everything in memory is treated as an object.*

This includes not just lists, strings, etc., but also less obvious things, such as

- functions (once they have been read into memory)
- modules (ditto)
- files opened for reading or writing
- integers, etc.

Consider, for example, functions.

When Python reads a function definition, it creates a **function object** and stores it in memory.

The following code illustrates

```
In [18]: def f(x): return x**2
         f
```

```
Out[18]: <function __main__.f(x)>
```

```
In [19]: type(f)
```

```
Out[19]: function
```

```
In [20]: id(f)
```

```
Out[20]: 139721126599536
```

```
In [21]: f.__name__
```

```
Out[21]: 'f'
```

We can see that `f` has type, identity, attributes and so on—just like any other object.

It also has methods.

One example is the `__call__` method, which just evaluates the function

```
In [22]: f.__call__(3)
```

```
Out[22]: 9
```

Another is the `__dir__` method, which returns a list of attributes.

Modules loaded into memory are also treated as objects

```
In [23]: import math

         id(math)
```

```
Out[23]: 139721223528840
```

This uniform treatment of data in Python (everything is an object) helps keep the language simple and consistent.

Chapter 7

OOP II: Building Classes

7.1 Contents

- Overview 7.2
- OOP Review 7.3
- Defining Your Own Classes 7.4
- Special Methods 7.5
- Exercises 7.6
- Solutions 7.7

7.2 Overview

In an earlier lecture, we learned some foundations of object-oriented programming.

The objectives of this lecture are

- cover OOP in more depth
- learn how to build our own objects, specialized to our needs

For example, you already know how to

- create lists, strings and other Python objects
- use their methods to modify their contents

So imagine now you want to write a program with consumers, who can

- hold and spend cash
- consume goods
- work and earn cash

A natural solution in Python would be to create consumers as objects with

- data, such as cash on hand
- methods, such as `buy` or `work` that affect this data

Python makes it easy to do this, by providing you with **class definitions**.

Classes are blueprints that help you build objects according to your own specifications.

It takes a little while to get used to the syntax so we'll provide plenty of examples.

We'll use the following imports:

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

7.3 OOP Review

OOP is supported in many languages:

- JAVA and Ruby are relatively pure OOP.
- Python supports both procedural and object-oriented programming.
- Fortran and MATLAB are mainly procedural, some OOP recently tacked on.
- C is a procedural language, while C++ is C with OOP added on top.

Let's cover general OOP concepts before we specialize to Python.

7.3.1 Key Concepts

As discussed an earlier lecture, in the OOP paradigm, data and functions are **bundled together** into "objects".

An example is a Python list, which not only stores data but also knows how to sort itself, etc.

```
In [2]: x = [1, 5, 4]
        x.sort()
        x
```

```
Out[2]: [1, 4, 5]
```

As we now know, `sort` is a function that is "part of" the list object — and hence called a *method*.

If we want to make our own types of objects we need to use class definitions.

A *class definition* is a blueprint for a particular class of objects (e.g., lists, strings or complex numbers).

It describes

- What kind of data the class stores
- What methods it has for acting on these data

An *object* or *instance* is a realization of the class, created from the blueprint

- Each instance has its own unique data.
- Methods set out in the class definition act on this (and other) data.

In Python, the data and methods of an object are collectively referred to as *attributes*.

Attributes are accessed via "dotted attribute notation"

- `object_name.data`
- `object_name.method_name()`

In the example

```
In [3]: x = [1, 5, 4]
        x.sort()
        x.__class__
```

```
Out[3]: list
```

- `x` is an object or instance, created from the definition for Python lists, but with its own particular data.
- `x.sort()` and `x.__class__` are two attributes of `x`.
- `dir(x)` can be used to view all the attributes of `x`.

7.3.2 Why is OOP Useful?

OOP is useful for the same reason that abstraction is useful: for recognizing and exploiting the common structure.

For example,

- *a Markov chain* consists of a set of states and a collection of transition probabilities for moving across states
- *a general equilibrium theory* consists of a commodity space, preferences, technologies, and an equilibrium definition
- *a game* consists of a list of players, lists of actions available to each player, player payoffs as functions of all players' actions, and a timing protocol

These are all abstractions that collect together "objects" of the same "type".

Recognizing common structure allows us to employ common tools.

In economic theory, this might be a proposition that applies to all games of a certain type.

In Python, this might be a method that's useful for all Markov chains (e.g., `simulate`).

When we use OOP, the `simulate` method is conveniently bundled together with the Markov chain object.

7.4 Defining Your Own Classes

Let's build some simple classes to start off.

7.4.1 Example: A Consumer Class

First, we'll build a `Consumer` class with

- a `wealth` attribute that stores the consumer's wealth (data)
- an `earn` method, where `earn(y)` increments the consumer's wealth by `y`
- a `spend` method, where `spend(x)` either decreases wealth by `x` or returns an error if insufficient funds exist

Admittedly a little contrived, this example of a class helps us internalize some new syntax.

Here's one implementation

```
In [4]: class Consumer:

            def __init__(self, w):
                "Initialize consumer with w dollars of wealth"
                self.wealth = w
```

```
    def earn(self, y):
        "The consumer earns y dollars"
        self.wealth += y

    def spend(self, x):
        "The consumer spends x dollars if feasible"
        new_wealth = self.wealth - x
        if new_wealth < 0:
            print("Insufficent funds")
        else:
            self.wealth = new_wealth
```

There's some special syntax here so let's step through carefully

- The `class` keyword indicates that we are building a class.

This class defines instance data `wealth` and three methods: `__init__`, `earn` and `spend`

- `wealth` is *instance data* because each consumer we create (each instance of the `Consumer` class) will have its own separate wealth data.

The ideas behind the `earn` and `spend` methods were discussed above.

Both of these act on the instance data `wealth`.

The `__init__` method is a *constructor method*.

Whenever we create an instance of the class, this method will be called automatically.

Calling `__init__` sets up a "namespace" to hold the instance data — more on this soon.

We'll also discuss the role of `self` just below.

Usage

Here's an example of usage

```
In [5]: c1 = Consumer(10)  # Create instance with initial wealth 10
        c1.spend(5)
        c1.wealth
```

```
Out[5]: 5
```

```
In [6]: c1.earn(15)
        c1.spend(100)
```

```
Insufficent funds
```

We can of course create multiple instances each with its own data

```
In [7]: c1 = Consumer(10)
        c2 = Consumer(12)
        c2.spend(4)
        c2.wealth
```

```
Out[7]: 8
```

```
In [8]: c1.wealth
```

```
Out[8]: 10
```

In fact, each instance stores its data in a separate namespace dictionary

```
In [9]: c1.__dict__
```

```
Out[9]: {'wealth': 10}
```

```
In [10]: c2.__dict__
```

```
Out[10]: {'wealth': 8}
```

When we access or set attributes we're actually just modifying the dictionary maintained by the instance.

Self

If you look at the `Consumer` class definition again you'll see the word self throughout the code.

The rules with `self` are that

- Any instance data should be prepended with `self`
  - e.g., the `earn` method references `self.wealth` rather than just `wealth`
- Any method defined within the class should have `self` as its first argument
  - e.g., `def earn(self, y)` rather than just `def earn(y)`
- Any method referenced within the class should be called as `self.method_name`

There are no examples of the last rule in the preceding code but we will see some shortly.

Details

In this section, we look at some more formal details related to classes and `self`

- You might wish to skip to the next section on first pass of this lecture.
- You can return to these details after you've familiarized yourself with more examples.

Methods actually live inside a class object formed when the interpreter reads the class definition

```
In [11]: print(Consumer.__dict__)  # Show __dict__ attribute of class object
```

```
    {'__module__': '__main__', '__init__': <function Consumer.__init__ at
 ↪0x7fbd2fa14378>,
'earn': <function Consumer.earn at 0x7fbd2fa142f0>, 'spend': <function Consumer.
 ↪spend
at 0x7fbd2fa14598>, '__dict__': <attribute '__dict__' of 'Consumer' objects>,
'__weakref__': <attribute '__weakref__' of 'Consumer' objects>, '__doc__': None}
```

Note how the three methods `__init__`, `earn` and `spend` are stored in the class object.

Consider the following code

```
In [12]: c1 = Consumer(10)
         c1.earn(10)
         c1.wealth
```

```
Out[12]: 20
```

When you call `earn` via `c1.earn(10)` the interpreter passes the instance `c1` and the argument `10` to `Consumer.earn`.

In fact, the following are equivalent

- `c1.earn(10)`
- `Consumer.earn(c1, 10)`

In the function call `Consumer.earn(c1, 10)` note that `c1` is the first argument.

Recall that in the definition of the `earn` method, `self` is the first parameter

```
In [13]: def earn(self, y):
             "The consumer earns y dollars"
             self.wealth += y
```

The end result is that `self` is bound to the instance `c1` inside the function call.

That's why the statement `self.wealth += y` inside `earn` ends up modifying `c1.wealth`.

7.4.2 Example: The Solow Growth Model

For our next example, let's write a simple class to implement the Solow growth model.

The Solow growth model is a neoclassical growth model where the amount of capital stock per capita k_t evolves according to the rule

$$k_{t+1} = \frac{szk_t^\alpha + (1-\delta)k_t}{1+n} \tag{1}$$

Here

- s is an exogenously given savings rate
- z is a productivity parameter
- α is capital's share of income
- n is the population growth rate
- δ is the depreciation rate

The **steady state** of the model is the k that solves (1) when $k_{t+1} = k_t = k$.

Here's a class that implements this model.

Some points of interest in the code are

- An instance maintains a record of its current capital stock in the variable `self.k`.
- The `h` method implements the right-hand side of (1).
- The `update` method uses `h` to update capital as per (1).

– Notice how inside `update` the reference to the local method `h` is `self.h`.

The methods `steady_state` and `generate_sequence` are fairly self-explanatory

```
In [14]: class Solow:
             r"""
             Implements the Solow growth model with the update rule

                 k_{t+1} = [(s z k^α_t) + (1 - δ)k_t] /(1 + n)

             """
             def __init__(self, n=0.05,  # population growth rate
                                s=0.25,  # savings rate
                                δ=0.1,   # depreciation rate
                                α=0.3,   # share of labor
                                z=2.0,   # productivity
                                k=1.0):  # current capital stock

                 self.n, self.s, self.δ, self.α, self.z = n, s, δ, α, z
                 self.k = k

             def h(self):
                 "Evaluate the h function"
                 # Unpack parameters (get rid of self to simplify notation)
                 n, s, δ, α, z = self.n, self.s, self.δ, self.α, self.z
                 # Apply the update rule
                 return (s * z * self.k**α + (1 - δ) * self.k) / (1 + n)

             def update(self):
                 "Update the current state (i.e., the capital stock)."
                 self.k =  self.h()

             def steady_state(self):
                 "Compute the steady state value of capital."
                 # Unpack parameters (get rid of self to simplify notation)
                 n, s, δ, α, z = self.n, self.s, self.δ, self.α, self.z
                 # Compute and return steady state
                 return ((s * z) / (n + δ))**(1 / (1 - α))

             def generate_sequence(self, t):
                 "Generate and return a time series of length t"
                 path = []
                 for i in range(t):
                     path.append(self.k)
                     self.update()
                 return path
```

Here's a little program that uses the class to compute time series from two different initial conditions.

The common steady state is also plotted for comparison

```
In [15]: s1 = Solow()
         s2 = Solow(k=8.0)

         T = 60
         fig, ax = plt.subplots(figsize=(9, 6))
```

```
# Plot the common steady state value of capital
ax.plot([s1.steady_state()]*T, 'k-', label='steady state')

# Plot time series for each economy
for s in s1, s2:
    lb = f'capital series from initial state {s.k}'
    ax.plot(s.generate_sequence(T), 'o-', lw=2, alpha=0.6, label=lb)

ax.set_xlabel('$k_{t+1}$', fontsize=14)
ax.set_ylabel('$k_t$', fontsize=14)
ax.legend()
plt.show()
```

7.4.3 Example: A Market

Next, let's write a class for a simple one good market where agents are price takers.

The market consists of the following objects:

- A linear demand curve $Q = a_d - b_d p$
- A linear supply curve $Q = a_z + b_z(p - t)$

Here

- p is price paid by the consumer, Q is quantity and t is a per-unit tax.
- Other symbols are demand and supply parameters.

The class provides methods to compute various values of interest, including competitive equilibrium price and quantity, tax revenue raised, consumer surplus and producer surplus.

Here's our implementation.

(It uses a function from SciPy called quad for numerical integration—a topic we will say more about later on.)

```
In [16]: from scipy.integrate import quad

         class Market:

             def __init__(self, ad, bd, az, bz, tax):
                 """
                 Set up market parameters.  All parameters are scalars.  See
                 https://lectures.quantecon.org/py/python_oop.html for
 ↪interpretation.

                 """
                 self.ad, self.bd, self.az, self.bz, self.tax = ad, bd, az, bz, tax
                 if ad < az:
                     raise ValueError('Insufficient demand.')

             def price(self):
                 "Return equilibrium price"
                return  (self.ad - self.az + self.bz * self.tax) / (self.bd + self.
 ↪bz)

             def quantity(self):
                 "Compute equilibrium quantity"
                 return  self.ad - self.bd * self.price()

             def consumer_surp(self):
                 "Compute consumer surplus"
                 # == Compute area under inverse demand function == #
                 integrand = lambda x: (self.ad / self.bd) - (1 / self.bd) * x
                 area, error = quad(integrand, 0, self.quantity())
                 return area - self.price() * self.quantity()

             def producer_surp(self):
                 "Compute producer surplus"
                 #  == Compute area above inverse supply curve, excluding tax == #
                 integrand = lambda x: -(self.az / self.bz) + (1 / self.bz) * x
                 area, error = quad(integrand, 0, self.quantity())
                 return (self.price() - self.tax) * self.quantity() - area

             def taxrev(self):
                 "Compute tax revenue"
                 return self.tax * self.quantity()

             def inverse_demand(self, x):
                 "Compute inverse demand"
                 return self.ad / self.bd - (1 / self.bd)* x

             def inverse_supply(self, x):
                 "Compute inverse supply curve"
                 return -(self.az / self.bz) + (1 / self.bz) * x + self.tax

             def inverse_supply_no_tax(self, x):
                 "Compute inverse supply curve without tax"
                 return -(self.az / self.bz) + (1 / self.bz) * x
```

Here's a sample of usage

In [17]:
```
baseline_params = 15, .5, -2, .5, 3
m = Market(*baseline_params)
print("equilibrium price = ", m.price())
```

```
equilibrium price =  18.5
```

In [18]:
```
print("consumer surplus = ", m.consumer_surp())
```

```
consumer surplus =  33.0625
```

Here's a short program that uses this class to plot an inverse demand curve together with inverse supply curves with and without taxes

In [19]:
```
# Baseline ad, bd, az, bz, tax
baseline_params = 15, .5, -2, .5, 3
m = Market(*baseline_params)

q_max = m.quantity() * 2
q_grid = np.linspace(0.0, q_max, 100)
pd = m.inverse_demand(q_grid)
ps = m.inverse_supply(q_grid)
psno = m.inverse_supply_no_tax(q_grid)

fig, ax = plt.subplots()
ax.plot(q_grid, pd, lw=2, alpha=0.6, label='demand')
ax.plot(q_grid, ps, lw=2, alpha=0.6, label='supply')
ax.plot(q_grid, psno, '--k', lw=2, alpha=0.6, label='supply without tax')
ax.set_xlabel('quantity', fontsize=14)
ax.set_xlim(0, q_max)
ax.set_ylabel('price', fontsize=14)
ax.legend(loc='lower right', frameon=False, fontsize=14)
plt.show()
```

The next program provides a function that

- takes an instance of `Market` as a parameter
- computes dead weight loss from the imposition of the tax

```
In [20]: def deadw(m):
             "Computes deadweight loss for market m."
             # == Create analogous market with no tax == #
             m_no_tax = Market(m.ad, m.bd, m.az, m.bz, 0)
             # == Compare surplus, return difference == #
             surp1 = m_no_tax.consumer_surp() + m_no_tax.producer_surp()
             surp2 = m.consumer_surp() + m.producer_surp() + m.taxrev()
             return surp1 - surp2
```

Here's an example of usage

```
In [21]: baseline_params = 15, .5, -2, .5, 3
         m = Market(*baseline_params)
         deadw(m)  # Show deadweight loss

Out[21]: 1.125
```

7.4.4 Example: Chaos

Let's look at one more example, related to chaotic dynamics in nonlinear systems.

One simple transition rule that can generate complex dynamics is the logistic map

$$x_{t+1} = rx_t(1 - x_t), \quad x_0 \in [0, 1], \quad r \in [0, 4] \tag{2}$$

Let's write a class for generating time series from this model.

Here's one implementation

```
In [22]: class Chaos:
             """
             Models the dynamical system with :math:`x_{t+1} = r x_t (1 - x_t)`
             """
             def __init__(self, x0, r):
                 """
                 Initialize with state x0 and parameter r
                 """
                 self.x, self.r = x0, r

             def update(self):
                 "Apply the map to update state."
                 self.x =  self.r * self.x *(1 - self.x)

             def generate_sequence(self, n):
                 "Generate and return a sequence of length n."
                 path = []
                 for i in range(n):
                     path.append(self.x)
                     self.update()
                 return path
```

Here's an example of usage

```
In [23]: ch = Chaos(0.1, 4.0)      # x0 = 0.1 and r = 0.4
         ch.generate_sequence(5)  # First 5 iterates
```

```
Out[23]: [0.1, 0.36000000000000004, 0.9216, 0.28901376000000006, 0.
 ↪8219392261226498]
```

This piece of code plots a longer trajectory

```
In [24]: ch = Chaos(0.1, 4.0)
         ts_length = 250

         fig, ax = plt.subplots()
         ax.set_xlabel('$t$', fontsize=14)
         ax.set_ylabel('$x_t$', fontsize=14)
         x = ch.generate_sequence(ts_length)
         ax.plot(range(ts_length), x, 'bo-', alpha=0.5, lw=2, label='$x_t$')
         plt.show()
```

The next piece of code provides a bifurcation diagram

```
In [25]: fig, ax = plt.subplots()
         ch = Chaos(0.1, 4)
         r = 2.5
         while r < 4:
             ch.r = r
             t = ch.generate_sequence(1000)[950:]
             ax.plot([r] * len(t), t, 'b.', ms=0.6)
             r = r + 0.005

         ax.set_xlabel('$r$', fontsize=16)
         ax.set_ylabel('$x_t$', fontsize=16)
         plt.show()
```

On the horizontal axis is the parameter r in (2).

The vertical axis is the state space $[0, 1]$.

For each r we compute a long time series and then plot the tail (the last 50 points).

The tail of the sequence shows us where the trajectory concentrates after settling down to some kind of steady state, if a steady state exists.

Whether it settles down, and the character of the steady state to which it does settle down, depend on the value of r.

For r between about 2.5 and 3, the time series settles into a single fixed point plotted on the vertical axis.

For r between about 3 and 3.45, the time series settles down to oscillating between the two values plotted on the vertical axis.

For r a little bit higher than 3.45, the time series settles down to oscillating among the four values plotted on the vertical axis.

Notice that there is no value of r that leads to a steady state oscillating among three values.

7.5 Special Methods

Python provides special methods with which some neat tricks can be performed.

For example, recall that lists and tuples have a notion of length and that this length can be queried via the `len` function

```
In [26]: x = (10, 20)
         len(x)
```

Out[26]: 2

If you want to provide a return value for the `len` function when applied to your user-defined object, use the `__len__` special method

```
In [27]: class Foo:

             def __len__(self):
                 return 42
```

Now we get

```
In [28]: f = Foo()
         len(f)
```

Out[28]: 42

A special method we will use regularly is the `__call__` method.

This method can be used to make your instances callable, just like functions

```
In [29]: class Foo:

             def __call__(self, x):
                 return x + 42
```

After running we get

```
In [30]: f = Foo()
         f(8)  # Exactly equivalent to f.__call__(8)
```

Out[30]: 50

Exercise 1 provides a more useful example.

7.6 Exercises

7.6.1 Exercise 1

The empirical cumulative distribution function (ecdf) corresponding to a sample $\{X_i\}_{i=1}^n$ is defined as

$$F_n(x) := \frac{1}{n} \sum_{i=1}^n \mathbf{1}\{X_i \leq x\} \qquad (x \in \mathbb{R}) \tag{3}$$

Here $\mathbf{1}\{X_i \leq x\}$ is an indicator function (one if $X_i \leq x$ and zero otherwise) and hence $F_n(x)$ is the fraction of the sample that falls below x.

The Glivenko–Cantelli Theorem states that, provided that the sample is IID, the ecdf F_n converges to the true distribution function F.

Implement F_n as a class called `ECDF`, where

- A given sample $\{X_i\}_{i=1}^n$ are the instance data, stored as `self.observations`.
- The class implements a `__call__` method that returns $F_n(x)$ for any x.

Your code should work as follows (modulo randomness)

```
from random import uniform

samples = [uniform(0, 1) for i in range(10)]
F = ECDF(samples)
F(0.5)  # Evaluate ecdf at x = 0.5

F.observations = [uniform(0, 1) for i in range(1000)]
F(0.5)
```

Aim for clarity, not efficiency.

7.6.2 Exercise 2

In an earlier exercise, you wrote a function for evaluating polynomials.

This exercise is an extension, where the task is to build a simple class called `Polynomial` for representing and manipulating polynomial functions such as

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_N x^N = \sum_{n=0}^{N} a_n x^n \qquad (x \in \mathbb{R}) \tag{4}$$

The instance data for the class `Polynomial` will be the coefficients (in the case of (4), the numbers $a_0, \dots, a_N$).

Provide methods that

1. Evaluate the polynomial (4), returning $p(x)$ for any x.
2. Differentiate the polynomial, replacing the original coefficients with those of its derivative p'.

Avoid using any `import` statements.

7.7 Solutions

7.7.1 Exercise 1

```
In [31]: class ECDF:

             def __init__(self, observations):
                 self.observations = observations

             def __call__(self, x):
                 counter = 0.0
                 for obs in self.observations:
```

```
            if obs <= x:
                counter += 1
        return counter / len(self.observations)
```

In [32]:
```
# == test == #

from random import uniform

samples = [uniform(0, 1) for i in range(10)]
F = ECDF(samples)

print(F(0.5))  # Evaluate ecdf at x = 0.5

F.observations = [uniform(0, 1) for i in range(1000)]

print(F(0.5))
```

```
0.6
0.52
```

7.7.2 Exercise 2

In [33]:
```
class Polynomial:

    def __init__(self, coefficients):
        """
        Creates an instance of the Polynomial class representing

            p(x) = a_0 x^0 + ... + a_N x^N,

        where a_i = coefficients[i].
        """
        self.coefficients = coefficients

    def __call__(self, x):
        "Evaluate the polynomial at x."
        y = 0
        for i, a in enumerate(self.coefficients):
            y += a * x**i
        return y

    def differentiate(self):
        "Reset self.coefficients to those of p' instead of p."
        new_coefficients = []
        for i, a in enumerate(self.coefficients):
            new_coefficients.append(i * a)
        # Remove the first element, which is zero
        del new_coefficients[0]
        # And reset coefficients data to new values
        self.coefficients = new_coefficients
        return new_coefficients
```

Part II

The Scientific Libraries

Chapter 8

Python for Scientific Computing

8.1 Contents

- Overview 8.2
- Scientific Libraries 8.3
- The Need for Speed 8.4
- Vectorization 8.5
- Beyond Vectorization 8.6

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

```
In [1]: !pip install --upgrade quantecon
```

8.2 Overview

Python is extremely popular for scientific computing, due to such factors as

- the accessible and flexible nature of the language itself,
- the huge range of high quality scientific libraries now available,
- the fact that the language and libraries are open source,
- the popular Anaconda Python distribution, which simplifies installation and management of those libraries, and
- the recent surge of interest in using Python for machine learning and artificial intelligence.

In this lecture we give a short overview of scientific computing in Python, addressing the following questions:

- What are the relative strengths and weaknesses of Python for these tasks?
- What are the main elements of the scientific Python ecosystem?
- How is the situation changing over time?

8.3 Scientific Libraries

Let's briefly review Python's scientific libraries, starting with why we need them.

8.3.1 The Role of Scientific Libraries

One obvious reason we use scientific libraries is because they implement routines we want to use.

For example, it's almost always better to use an existing routine for root finding than to write a new one from scratch.

(For standard algorithms, efficiency is maximized if the community can coordinate on a common set of implementations, written by experts and tuned by users to be as fast and robust as possible.)

But this is not the only reason that we use Python's scientific libraries.

Another is that pure Python, while flexible and elegant, is not fast.

So we need libraries that are designed to accelerate execution of Python code.

As we'll see below, there are now Python libraries that can do this extremely well.

8.3.2 Python's Scientific Ecosystem

In terms of popularity, the big four in the world of scientific Python libraries are

- NumPy
- SciPy
- Matplotlib
- Pandas

For us, there's another (relatively new) library that will also be essential for numerical computing:

- Numba

Over the next few lectures we'll see how to use these libraries.

But first, let's quickly review how they fit together.

- NumPy forms the foundations by providing a basic array data type (think of vectors and matrices) and functions for acting on these arrays (e.g., matrix multiplication).
- SciPy builds on NumPy by adding the kinds of numerical methods that are routinely used in science (interpolation, optimization, root finding, etc.).
- Matplotlib is used to generate figures, with a focus on plotting data stored in NumPy arrays.
- Pandas provides types and functions for empirical work (e.g., manipulating data).
- Numba accelerates execution via JIT compilation — we'll learn about this soon.

8.4 The Need for Speed

Now let's discuss execution speed.

Higher-level languages like Python are optimized for humans.

This means that the programmer can leave many details to the runtime environment

- specifying variable types
- memory allocation/deallocation, etc.

The upside is that, compared to low-level languages, Python is typically faster to write, less error-prone and easier to debug.

The downside is that Python is harder to optimize — that is, turn into fast machine code — than languages like C or Fortran.

Indeed, the standard implementation of Python (called CPython) cannot match the speed of compiled languages such as C or Fortran.

Does that mean that we should just switch to C or Fortran for everything?

The answer is: No, no and one hundred times no!

(This is what you should say to the senior professor insisting that the model needs to be rewritten in Fortran or C++.)

There are two reasons why:

First, for any given program, relatively few lines are ever going to be time-critical.

Hence it is far more efficient to write most of our code in a high productivity language like Python.

Second, even for those lines of code that *are* time-critical, we can now achieve the same speed as C or Fortran using Python's scientific libraries.

8.4.1 Where are the Bottlenecks?

Before we learn how to do this, let's try to understand why plain vanilla Python is slower than C or Fortran.

This will, in turn, help us figure out how to speed things up.

Dynamic Typing

Consider this Python operation

```
In [2]: a, b = 10, 10
        a + b
```

```
Out[2]: 20
```

Even for this simple operation, the Python interpreter has a fair bit of work to do.

For example, in the statement `a + b`, the interpreter has to know which operation to invoke.

If `a` and `b` are strings, then `a + b` requires string concatenation

```
In [3]: a, b = 'foo', 'bar'
        a + b
```

```
Out[3]: 'foobar'
```

If `a` and `b` are lists, then `a + b` requires list concatenation

```
In [4]: a, b = ['foo'], ['bar']
        a + b
```

```
Out[4]: ['foo', 'bar']
```

(We say that the operator `+` is *overloaded* — its action depends on the type of the objects on which it acts)

As a result, Python must check the type of the objects and then call the correct operation.

This involves substantial overheads.

Static Types

Compiled languages avoid these overheads with explicit, static types.

For example, consider the following C code, which sums the integers from 1 to 10

```
#include <stdio.h>

int main(void) {
    int i;
    int sum = 0;
    for (i = 1; i <= 10; i++) {
        sum = sum + i;
    }
    printf("sum = %d\n", sum);
    return 0;
}
```

The variables `i` and `sum` are explicitly declared to be integers.

Hence, the meaning of addition here is completely unambiguous.

8.4.2 Data Access

Another drag on speed for high-level languages is data access.

To illustrate, let's consider the problem of summing some data — say, a collection of integers.

Summing with Compiled Code

In C or Fortran, these integers would typically be stored in an array, which is a simple data structure for storing homogeneous data.

Such an array is stored in a single contiguous block of memory

- In modern computers, memory addresses are allocated to each byte (one byte = 8 bits).
- For example, a 64 bit integer is stored in 8 bytes of memory.
- An array of n such integers occupies $8n$ **consecutive** memory slots.

Moreover, the compiler is made aware of the data type by the programmer.

- In this case 64 bit integers

Hence, each successive data point can be accessed by shifting forward in memory space by a known and fixed amount.

- In this case 8 bytes

Summing in Pure Python

Python tries to replicate these ideas to some degree.

For example, in the standard Python implementation (CPython), list elements are placed in memory locations that are in a sense contiguous.

However, these list elements are more like pointers to data rather than actual data.

Hence, there is still overhead involved in accessing the data values themselves.

This is a considerable drag on speed.

In fact, it's generally true that memory traffic is a major culprit when it comes to slow execution.

Let's look at some ways around these problems.

8.5 Vectorization

There is a clever method called **vectorization** that can be used to speed up high level languages in numerical applications.

The key idea is to send array processing operations in batch to pre-compiled and efficient native machine code.

The machine code itself is typically compiled from carefully optimized C or Fortran.

For example, when working in a high level language, the operation of inverting a large matrix can be subcontracted to efficient machine code that is pre-compiled for this purpose and supplied to users as part of a package.

This clever idea dates back to MATLAB, which uses vectorization extensively.

Vectorization can greatly accelerate many numerical computations (but not all, as we shall see).

Let's see how vectorization works in Python, using NumPy.

8.5.1 Operations on Arrays

First, let's run some imports

```
In [5]: import random
        import numpy as np
        import quantecon as qe
```

Next let's try some non-vectorized code, which uses a native Python loop to generate, square and then sum a large number of random variables:

```
In [6]: n = 1_000_000
```

In [7]:
```
%%time

y = 0      # Will accumulate and store sum
for i in range(n):
    x = random.uniform(0, 1)
    y += x**2
```

```
   CPU times: user 719 ms, sys: 0 ns, total: 719 ms
Wall time: 719 ms
```

The following vectorized code achieves the same thing.

In [8]:
```
%%time

x = np.random.uniform(0, 1, n)
y = np.sum(x**2)
```

```
   CPU times: user 25.9 ms, sys: 4.14 ms, total: 30 ms
Wall time: 38.9 ms
```

As you can see, the second code block runs much faster. Why?

The second code block breaks the loop down into three basic operations

1. draw `n` uniforms
2. square them
3. sum them

These are sent as batch operators to optimized machine code.

Apart from minor overheads associated with sending data back and forth, the result is C or Fortran-like speed.

When we run batch operations on arrays like this, we say that the code is *vectorized*.

Vectorized code is typically fast and efficient.

It is also surprisingly flexible, in the sense that many operations can be vectorized.

The next section illustrates this point.

8.5.2 Universal Functions

Many functions provided by NumPy are so-called *universal functions* — also called ufuncs.

This means that they

- map scalars into scalars, as expected
- map arrays into arrays, acting element-wise

For example, `np.cos` is a ufunc:

In [9]:
```
np.cos(1.0)
```

Out[9]:
```
0.5403023058681398
```

In [10]:
```
np.cos(np.linspace(0, 1, 3))
```

Out[10]:
```
array([1.        , 0.87758256, 0.54030231])
```

By exploiting ufuncs, many operations can be vectorized.

For example, consider the problem of maximizing a function f of two variables (x, y) over the square $[-a, a] \times [-a, a]$.

For f and a let's choose

$$f(x, y) = \frac{\cos(x^2 + y^2)}{1 + x^2 + y^2} \quad \text{and} \quad a = 3$$

Here's a plot of f

In [11]:
```
import matplotlib.pyplot as plt
%matplotlib inline
from mpl_toolkits.mplot3d.axes3d import Axes3D
from matplotlib import cm

def f(x, y):
    return np.cos(x**2 + y**2) / (1 + x**2 + y**2)

xgrid = np.linspace(-3, 3, 50)
ygrid = xgrid
x, y = np.meshgrid(xgrid, ygrid)

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x,
                y,
                f(x, y),
                rstride=2, cstride=2,
                cmap=cm.jet,
                alpha=0.7,
                linewidth=0.25)
ax.set_zlim(-0.5, 1.0)
ax.set_xlabel('$x$', fontsize=14)
ax.set_ylabel('$y$', fontsize=14)
plt.show()
```

To maximize it, we're going to use a naive grid search:

1. Evaluate f for all (x, y) in a grid on the square.
2. Return the maximum of observed values.

The grid will be

```
In [12]: grid = np.linspace(-3, 3, 1000)
```

Here's a non-vectorized version that uses Python loops.

```
In [13]: %%time

         m = -np.inf

         for x in grid:
             for y in grid:
                 z = f(x, y)
                 if z > m:
                     m = z
```

```
    CPU times: user 4.72 s, sys: 0 ns, total: 4.72 s
Wall time: 4.88 s
```

And here's a vectorized version

In [14]:
```
%%time

x, y = np.meshgrid(grid, grid)
np.max(f(x, y))
```

```
CPU times: user 18.8 ms, sys: 19.7 ms, total: 38.5 ms
Wall time: 38.2 ms
```

Out[14]:
```
0.9999819641085747
```

In the vectorized version, all the looping takes place in compiled code.

As you can see, the second version is **much** faster.

(We'll make it even faster again later on, using more scientific programming tricks.)

8.6 Beyond Vectorization

At its best, vectorization yields fast, simple code.

However, it's not without disadvantages.

One issue is that it can be highly memory-intensive.

For example, the vectorized maximization routine above is far more memory intensive than the non-vectorized version that preceded it.

This is because vectorization tends to create many intermediate arrays before producing the final calculation.

Another issue is that not all algorithms can be vectorized.

In these kinds of settings, we need to go back to loops.

Fortunately, there are alternative ways to speed up Python loops that work in almost any setting.

For example, in the last few years, a new Python library called Numba has appeared that solves the main problems with vectorization listed above.

It does so through something called **just in time (JIT) compilation**, which can generate extremely fast and efficient code.

We'll learn how to use Numba soon.

Chapter 9

NumPy

9.1 Contents

- Overview 9.2
- NumPy Arrays 9.3
- Operations on Arrays 9.4
- Additional Functionality 9.5
- Exercises 9.6
- Solutions 9.7

> "Let's be clear: the work of science has nothing whatever to do with consensus. Consensus is the business of politics. Science, on the contrary, requires only one investigator who happens to be right, which means that he or she has results that are verifiable by reference to the real world. In science consensus is irrelevant. What is relevant is reproducible results." – Michael Crichton

9.2 Overview

NumPy is a first-rate library for numerical programming

- Widely used in academia, finance and industry.
- Mature, fast, stable and under continuous development.

We have already seen some code involving NumPy in the preceding lectures.

In this lecture, we will start a more systematic discussion of both

- NumPy arrays and
- the fundamental array processing operations provided by NumPy.

9.2.1 References

- The official NumPy documentation.

9.3 NumPy Arrays

The essential problem that NumPy solves is fast array processing.

The most important structure that NumPy defines is an array data type formally called a numpy.ndarray.

NumPy arrays power a large proportion of the scientific Python ecosystem.

Let's first import the library.

```
In [1]: import numpy as np
```

To create a NumPy array containing only zeros we use np.zeros

```
In [2]: a = np.zeros(3)
        a
```

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

```
In [3]: type(a)
```

```
Out[3]: numpy.ndarray
```

NumPy arrays are somewhat like native Python lists, except that

- Data *must be homogeneous* (all elements of the same type).
- These types must be one of the data types (`dtypes`) provided by NumPy.

The most important of these dtypes are:

- float64: 64 bit floating-point number
- int64: 64 bit integer
- bool: 8 bit True or False

There are also dtypes to represent complex numbers, unsigned integers, etc.

On modern machines, the default dtype for arrays is `float64`

```
In [4]: a = np.zeros(3)
        type(a[0])
```

```
Out[4]: numpy.float64
```

If we want to use integers we can specify as follows:

```
In [5]: a = np.zeros(3, dtype=int)
        type(a[0])
```

```
Out[5]: numpy.int64
```

9.3.1 Shape and Dimension

Consider the following assignment

```
In [6]: z = np.zeros(10)
```

Here `z` is a *flat* array with no dimension — neither row nor column vector.

The dimension is recorded in the `shape` attribute, which is a tuple

```
In [7]: z.shape

Out[7]: (10,)
```

Here the shape tuple has only one element, which is the length of the array (tuples with one element end with a comma).

To give it dimension, we can change the `shape` attribute

```
In [8]: z.shape = (10, 1)
        z

Out[8]: array([[0.],
               [0.],
               [0.],
               [0.],
               [0.],
               [0.],
               [0.],
               [0.],
               [0.],
               [0.]])

In [9]: z = np.zeros(4)
        z.shape = (2, 2)
        z

Out[9]: array([[0., 0.],
               [0., 0.]])
```

In the last case, to make the 2 by 2 array, we could also pass a tuple to the `zeros()` function, as in `z = np.zeros((2, 2))`.

9.3.2 Creating Arrays

As we've seen, the `np.zeros` function creates an array of zeros.

You can probably guess what `np.ones` creates.

Related is `np.empty`, which creates arrays in memory that can later be populated with data

```
In [10]: z = np.empty(3)
         z
```

```
Out[10]: array([0., 0., 0.])
```

The numbers you see here are garbage values.

(Python allocates 3 contiguous 64 bit pieces of memory, and the existing contents of those memory slots are interpreted as `float64` values)

To set up a grid of evenly spaced numbers use `np.linspace`

```
In [11]: z = np.linspace(2, 4, 5)  # From 2 to 4, with 5 elements
```

To create an identity matrix use either `np.identity` or `np.eye`

```
In [12]: z = np.identity(2)
         z

Out[12]: array([[1., 0.],
                [0., 1.]])
```

In addition, NumPy arrays can be created from Python lists, tuples, etc. using `np.array`

```
In [13]: z = np.array([10, 20])                 # ndarray from Python list
         z

Out[13]: array([10, 20])

In [14]: type(z)

Out[14]: numpy.ndarray

In [15]: z = np.array((10, 20), dtype=float)    # Here 'float' is equivalent to
 ↪'np.float64'
         z

Out[15]: array([10., 20.])

In [16]: z = np.array([[1, 2], [3, 4]])         # 2D array from a list of lists
         z

Out[16]: array([[1, 2],
                [3, 4]])
```

See also `np.asarray`, which performs a similar function, but does not make a distinct copy of data already in a NumPy array.

```
In [17]: na = np.linspace(10, 20, 2)
         na is np.asarray(na)   # Does not copy NumPy arrays

Out[17]: True

In [18]: na is np.array(na)     # Does make a new copy --- perhaps unnecessarily

Out[18]: False
```

To read in the array data from a text file containing numeric data use `np.loadtxt` or `np.genfromtxt`—see the documentation for details.

9.3.3 Array Indexing

For a flat array, indexing is the same as Python sequences:

```
In [19]: z = np.linspace(1, 2, 5)
         z

Out[19]: array([1.  , 1.25, 1.5 , 1.75, 2.  ])

In [20]: z[0]

Out[20]: 1.0

In [21]: z[0:2]  # Two elements, starting at element 0

Out[21]: array([1.  , 1.25])

In [22]: z[-1]

Out[22]: 2.0
```

For 2D arrays the index syntax is as follows:

```
In [23]: z = np.array([[1, 2], [3, 4]])
         z

Out[23]: array([[1, 2],
                [3, 4]])

In [24]: z[0, 0]

Out[24]: 1

In [25]: z[0, 1]

Out[25]: 2
```

And so on.

Note that indices are still zero-based, to maintain compatibility with Python sequences.

Columns and rows can be extracted as follows

```
In [26]: z[0, :]

Out[26]: array([1, 2])

In [27]: z[:, 1]
```

```
Out[27]: array([2, 4])
```

NumPy arrays of integers can also be used to extract elements

```
In [28]: z = np.linspace(2, 4, 5)
         z
```

```
Out[28]: array([2. , 2.5, 3. , 3.5, 4. ])
```

```
In [29]: indices = np.array((0, 2, 3))
         z[indices]
```

```
Out[29]: array([2. , 3. , 3.5])
```

Finally, an array of `dtype bool` can be used to extract elements

```
In [30]: z
```

```
Out[30]: array([2. , 2.5, 3. , 3.5, 4. ])
```

```
In [31]: d = np.array([0, 1, 1, 0, 0], dtype=bool)
         d
```

```
Out[31]: array([False,  True,  True, False, False])
```

```
In [32]: z[d]
```

```
Out[32]: array([2.5, 3. ])
```

We'll see why this is useful below.

An aside: all elements of an array can be set equal to one number using slice notation

```
In [33]: z = np.empty(3)
         z
```

```
Out[33]: array([2. , 3. , 3.5])
```

```
In [34]: z[:] = 42
         z
```

```
Out[34]: array([42., 42., 42.])
```

9.3.4 Array Methods

Arrays have useful methods, all of which are carefully optimized

```
In [35]: a = np.array((4, 3, 2, 1))
         a

Out[35]: array([4, 3, 2, 1])

In [36]: a.sort()              # Sorts a in place
         a

Out[36]: array([1, 2, 3, 4])

In [37]: a.sum()               # Sum

Out[37]: 10

In [38]: a.mean()              # Mean

Out[38]: 2.5

In [39]: a.max()               # Max

Out[39]: 4

In [40]: a.argmax()            # Returns the index of the maximal element

Out[40]: 3

In [41]: a.cumsum()            # Cumulative sum of the elements of a

Out[41]: array([ 1,  3,  6, 10])

In [42]: a.cumprod()           # Cumulative product of the elements of a

Out[42]: array([ 1,  2,  6, 24])

In [43]: a.var()               # Variance

Out[43]: 1.25

In [44]: a.std()               # Standard deviation

Out[44]: 1.118033988749895

In [45]: a.shape = (2, 2)
         a.T                   # Equivalent to a.transpose()
```

```
Out[45]: array([[1, 3],
                [2, 4]])
```

Another method worth knowing is `searchsorted()`.

If `z` is a nondecreasing array, then `z.searchsorted(a)` returns the index of the first element of `z` that is `>= a`

```
In [46]: z = np.linspace(2, 4, 5)
         z
```

```
Out[46]: array([2. , 2.5, 3. , 3.5, 4. ])
```

```
In [47]: z.searchsorted(2.2)
```

```
Out[47]: 1
```

Many of the methods discussed above have equivalent functions in the NumPy namespace

```
In [48]: a = np.array((4, 3, 2, 1))
```

```
In [49]: np.sum(a)
```

```
Out[49]: 10
```

```
In [50]: np.mean(a)
```

```
Out[50]: 2.5
```

9.4 Operations on Arrays

9.4.1 Arithmetic Operations

The operators `+`, `-`, `*`, `/` and `**` all act *elementwise* on arrays

```
In [51]: a = np.array([1, 2, 3, 4])
         b = np.array([5, 6, 7, 8])
         a + b
```

```
Out[51]: array([ 6,  8, 10, 12])
```

```
In [52]: a * b
```

```
Out[52]: array([ 5, 12, 21, 32])
```

We can add a scalar to each element as follows

```
In [53]: a + 10
```

```
Out[53]: array([11, 12, 13, 14])
```

Scalar multiplication is similar

```
In [54]: a * 10
```

```
Out[54]: array([10, 20, 30, 40])
```

The two-dimensional arrays follow the same general rules

```
In [55]: A = np.ones((2, 2))
         B = np.ones((2, 2))
         A + B
```

```
Out[55]: array([[2., 2.],
                [2., 2.]])
```

```
In [56]: A + 10
```

```
Out[56]: array([[11., 11.],
                [11., 11.]])
```

```
In [57]: A * B
```

```
Out[57]: array([[1., 1.],
                [1., 1.]])
```

In particular, `A * B` is *not* the matrix product, it is an element-wise product.

9.4.2 Matrix Multiplication

With Anaconda's scientific Python package based around Python 3.5 and above, one can use the `@` symbol for matrix multiplication, as follows:

```
In [58]: A = np.ones((2, 2))
         B = np.ones((2, 2))
         A @ B
```

```
Out[58]: array([[2., 2.],
                [2., 2.]])
```

(For older versions of Python and NumPy you need to use the np.dot function)

We can also use `@` to take the inner product of two flat arrays

```
In [59]: A = np.array((1, 2))
         B = np.array((10, 20))
         A @ B
```

```
Out[59]: 50
```

In fact, we can use @ when one element is a Python list or tuple

```
In [60]: A = np.array(((1, 2), (3, 4)))
         A
```

```
Out[60]: array([[1, 2],
                [3, 4]])
```

```
In [61]: A @ (0, 1)
```

```
Out[61]: array([2, 4])
```

Since we are post-multiplying, the tuple is treated as a column vector.

9.4.3 Mutability and Copying Arrays

NumPy arrays are mutable data types, like Python lists.

In other words, their contents can be altered (mutated) in memory after initialization.

We already saw examples above.

Here's another example:

```
In [62]: a = np.array([42, 44])
         a
```

```
Out[62]: array([42, 44])
```

```
In [63]: a[-1] = 0  # Change last element to 0
         a
```

```
Out[63]: array([42,  0])
```

Mutability leads to the following behavior (which can be shocking to MATLAB programmers…)

```
In [64]: a = np.random.randn(3)
         a
```

```
Out[64]: array([-1.16873149,  0.16185124, -0.04144057])
```

```
In [65]: b = a
         b[0] = 0.0
         a
```

```
Out[65]: array([ 0.        ,  0.16185124, -0.04144057])
```

What's happened is that we have changed `a` by changing `b`.

The name `b` is bound to `a` and becomes just another reference to the array (the Python assignment model is described in more detail later in the course).

Hence, it has equal rights to make changes to that array.

This is in fact the most sensible default behavior!

It means that we pass around only pointers to data, rather than making copies.

Making copies is expensive in terms of both speed and memory.

Making Copies

It is of course possible to make `b` an independent copy of `a` when required.

This can be done using `np.copy`

```
In [66]: a = np.random.randn(3)
         a
```

```
Out[66]: array([-1.38036073,  0.23404972,  0.17611812])
```

```
In [67]: b = np.copy(a)
         b
```

```
Out[67]: array([-1.38036073,  0.23404972,  0.17611812])
```

Now `b` is an independent copy (called a *deep copy*)

```
In [68]: b[:] = 1
         b
```

```
Out[68]: array([1., 1., 1.])
```

```
In [69]: a
```

```
Out[69]: array([-1.38036073,  0.23404972,  0.17611812])
```

Note that the change to `b` has not affected `a`.

9.5 Additional Functionality

Let's look at some other useful things we can do with NumPy.

9.5.1 Vectorized Functions

NumPy provides versions of the standard functions `log`, `exp`, `sin`, etc. that act *element-wise* on arrays

```
In [70]: z = np.array([1, 2, 3])
         np.sin(z)
```

```
Out[70]: array([0.84147098, 0.90929743, 0.14112001])
```

This eliminates the need for explicit element-by-element loops such as

```
In [71]: n = len(z)
         y = np.empty(n)
         for i in range(n):
             y[i] = np.sin(z[i])
```

Because they act element-wise on arrays, these functions are called *vectorized functions*.

In NumPy-speak, they are also called *ufuncs*, which stands for "universal functions".

As we saw above, the usual arithmetic operations (`+`, `*`, etc.) also work element-wise, and combining these with the ufuncs gives a very large set of fast element-wise functions.

```
In [72]: z
```

```
Out[72]: array([1, 2, 3])
```

```
In [73]: (1 / np.sqrt(2 * np.pi)) * np.exp(- 0.5 * z**2)
```

```
Out[73]: array([0.24197072, 0.05399097, 0.00443185])
```

Not all user-defined functions will act element-wise.

For example, passing the function `f` defined below a NumPy array causes a `ValueError`

```
In [74]: def f(x):
             return 1 if x > 0 else 0
```

The NumPy function `np.where` provides a vectorized alternative:

```
In [75]: x = np.random.randn(4)
         x
```

```
Out[75]: array([-0.84390494, -1.26112843,  0.63113513, -0.34345913])
```

```
In [76]: np.where(x > 0, 1, 0)  # Insert 1 if x > 0 true, otherwise 0
```

```
Out[76]: array([0, 0, 1, 0])
```

You can also use `np.vectorize` to vectorize a given function

```
In [77]: f = np.vectorize(f)
         f(x)                # Passing the same vector x as in the previous example
```

```
Out[77]: array([0, 0, 1, 0])
```

However, this approach doesn't always obtain the same speed as a more carefully crafted vectorized function.

9.5.2 Comparisons

As a rule, comparisons on arrays are done element-wise

```
In [78]: z = np.array([2, 3])
         y = np.array([2, 3])
         z == y

Out[78]: array([ True,  True])

In [79]: y[0] = 5
         z == y

Out[79]: array([False,  True])

In [80]: z != y

Out[80]: array([ True, False])
```

The situation is similar for `>`, `<`, `>=` and `<=`.

We can also do comparisons against scalars

```
In [81]: z = np.linspace(0, 10, 5)
         z

Out[81]: array([ 0. ,  2.5,  5. ,  7.5, 10. ])

In [82]: z > 3

Out[82]: array([False, False,  True,  True,  True])
```

This is particularly useful for *conditional extraction*

```
In [83]: b = z > 3
         b

Out[83]: array([False, False,  True,  True,  True])

In [84]: z[b]

Out[84]: array([ 5. ,  7.5, 10. ])
```

Of course we can—and frequently do—perform this in one step

```
In [85]: z[z > 3]

Out[85]: array([ 5. ,  7.5, 10. ])
```

9.5.3 Sub-packages

NumPy provides some additional functionality related to scientific programming through its sub-packages.

We've already seen how we can generate random variables using np.random

```
In [86]: z = np.random.randn(10000)  # Generate standard normals
         y = np.random.binomial(10, 0.5, size=1000)    # 1,000 draws from Bin(10,
 ↪0.5)
         y.mean()

Out[86]: 5.067
```

Another commonly used subpackage is np.linalg

```
In [87]: A = np.array([[1, 2], [3, 4]])

         np.linalg.det(A)           # Compute the determinant

Out[87]: -2.0000000000000004

In [88]: np.linalg.inv(A)           # Compute the inverse

Out[88]: array([[-2. ,  1. ],
                [ 1.5, -0.5]])
```

Much of this functionality is also available in SciPy, a collection of modules that are built on top of NumPy.

We'll cover the SciPy versions in more detail soon.

For a comprehensive list of what's available in NumPy see this documentation.

9.6 Exercises

9.6.1 Exercise 1

Consider the polynomial expression

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_N x^N = \sum_{n=0}^{N} a_n x^n \tag{1}$$

Earlier, you wrote a simple function `p(x, coeff)` to evaluate (1) without considering efficiency.

Now write a new function that does the same job, but uses NumPy arrays and array operations for its computations, rather than any form of Python loop.

(Such functionality is already implemented as `np.poly1d`, but for the sake of the exercise don't use this class)

- Hint: Use `np.cumprod()`

9.6.2 Exercise 2

Let `q` be a NumPy array of length `n` with `q.sum() == 1`.

Suppose that `q` represents a probability mass function.

We wish to generate a discrete random variable x such that $\mathbb{P}\{x = i\} = q_i$.

In other words, `x` takes values in `range(len(q))` and `x = i` with probability `q[i]`.

The standard (inverse transform) algorithm is as follows:

- Divide the unit interval $[0, 1]$ into n subintervals $I_0, I_1, \dots, I_{n-1}$ such that the length of I_i is q_i.
- Draw a uniform random variable U on $[0, 1]$ and return the i such that $U \in I_i$.

The probability of drawing i is the length of I_i, which is equal to q_i.

We can implement the algorithm as follows

In [89]:
```
from random import uniform

def sample(q):
    a = 0.0
    U = uniform(0, 1)
    for i in range(len(q)):
        if a < U <= a + q[i]:
            return i
        a = a + q[i]
```

If you can't see how this works, try thinking through the flow for a simple example, such as `q = [0.25, 0.75]` It helps to sketch the intervals on paper.

Your exercise is to speed it up using NumPy, avoiding explicit loops

- Hint: Use `np.searchsorted` and `np.cumsum`

If you can, implement the functionality as a class called `DiscreteRV`, where

- the data for an instance of the class is the vector of probabilities `q`
- the class has a `draw()` method, which returns one draw according to the algorithm described above

If you can, write the method so that `draw(k)` returns `k` draws from `q`.

9.6.3 Exercise 3

Recall our earlier discussion of the empirical cumulative distribution function.

Your task is to

1. Make the `__call__` method more efficient using NumPy.

2. Add a method that plots the ECDF over $[a, b]$, where a and b are method parameters.

9.7 Solutions

```
In [90]: import matplotlib.pyplot as plt
         %matplotlib inline
```

9.7.1 Exercise 1

This code does the job

```
In [91]: def p(x, coef):
             X = np.ones_like(coef)
             X[1:] = x
             y = np.cumprod(X)   # y = [1, x, x**2,...]
             return coef @ y
```

Let's test it

```
In [92]: x = 2
         coef = np.linspace(2, 4, 3)
         print(coef)
         print(p(x, coef))
         # For comparison
         q = np.poly1d(np.flip(coef))
         print(q(x))
```

```
[2. 3. 4.]
24.0
24.0
```

9.7.2 Exercise 2

Here's our first pass at a solution:

```
In [93]: from numpy import cumsum
         from numpy.random import uniform

         class DiscreteRV:
             """
             Generates an array of draws from a discrete random variable with
 ↪vector of
             probabilities given by q.
             """

             def __init__(self, q):
                 """
                The argument q is a NumPy array, or array like, nonnegative and sums
                 to 1
                 """
                 self.q = q
                 self.Q = cumsum(q)

             def draw(self, k=1):
```

```
        """
        Returns k draws from q. For each such draw, the value i is returned
         with probability q[i].
         """
         return self.Q.searchsorted(uniform(0, 1, size=k))
```

The logic is not obvious, but if you take your time and read it slowly, you will understand.

There is a problem here, however.

Suppose that `q` is altered after an instance of `discreteRV` is created, for example by

```
In [94]: q = (0.1, 0.9)
         d = DiscreteRV(q)
         d.q = (0.5, 0.5)
```

The problem is that `Q` does not change accordingly, and `Q` is the data used in the `draw` method.

To deal with this, one option is to compute `Q` every time the draw method is called.

But this is inefficient relative to computing `Q` once-off.

A better option is to use descriptors.

A solution from the quantecon library using descriptors that behaves as we desire can be found here.

9.7.3 Exercise 3

An example solution is given below.

In essence, we've just taken this code from QuantEcon and added in a plot method

```
In [95]: """
         Modifies ecdf.py from QuantEcon to add in a plot method

         """

         class ECDF:
             """
             One-dimensional empirical distribution function given a vector of
             observations.

             Parameters
             ----------
             observations : array_like
                 An array of observations

             Attributes
             ----------
             observations : array_like
                 An array of observations

             """

             def __init__(self, observations):
```

```
        self.observations = np.asarray(observations)

    def __call__(self, x):
        """
        Evaluates the ecdf at x

        Parameters
        ----------
        x : scalar(float)
            The x at which the ecdf is evaluated

        Returns
        -------
        scalar(float)
            Fraction of the sample less than x

        """
        return np.mean(self.observations <= x)

    def plot(self, ax, a=None, b=None):
        """
        Plot the ecdf on the interval [a, b].

        Parameters
        ----------
        a : scalar(float), optional(default=None)
            Lower endpoint of the plot interval
        b : scalar(float), optional(default=None)
            Upper endpoint of the plot interval

        """

        # === choose reasonable interval if [a, b] not specified === #
        if a is None:
            a = self.observations.min() - self.observations.std()
        if b is None:
            b = self.observations.max() + self.observations.std()

        # === generate plot === #
        x_vals = np.linspace(a, b, num=100)
        f = np.vectorize(self.__call__)
        ax.plot(x_vals, f(x_vals))
        plt.show()
```

Here's an example of usage

```
In [96]: fig, ax = plt.subplots()
         X = np.random.randn(1000)
         F = ECDF(X)
         F.plot(ax)
```

1.0
0.8
0.6
0.4
0.2
0.0
−4
−2
0
2
4

Chapter 10

Matplotlib

10.1 Contents

- Overview 10.2
- The APIs 10.3
- More Features 10.4
- Further Reading 10.5
- Exercises 10.6
- Solutions 10.7

10.2 Overview

We've already generated quite a few figures in these lectures using Matplotlib.

Matplotlib is an outstanding graphics library, designed for scientific computing, with

- high-quality 2D and 3D plots
- output in all the usual formats (PDF, PNG, etc.)
- LaTeX integration
- fine-grained control over all aspects of presentation
- animation, etc.

10.2.1 Matplotlib's Split Personality

Matplotlib is unusual in that it offers two different interfaces to plotting.

One is a simple MATLAB-style API (Application Programming Interface) that was written to help MATLAB refugees find a ready home.

The other is a more "Pythonic" object-oriented API.

For reasons described below, we recommend that you use the second API.

But first, let's discuss the difference.

10.3 The APIs

10.3.1 The MATLAB-style API

Here's the kind of easy example you might find in introductory treatments

```
In [1]: import matplotlib.pyplot as plt
        %matplotlib inline
        import numpy as np

        x = np.linspace(0, 10, 200)
        y = np.sin(x)

        plt.plot(x, y, 'b-', linewidth=2)
        plt.show()
```

This is simple and convenient, but also somewhat limited and un-Pythonic.

For example, in the function calls, a lot of objects get created and passed around without making themselves known to the programmer.

Python programmers tend to prefer a more explicit style of programming (run `import this` in a code block and look at the second line).

This leads us to the alternative, object-oriented Matplotlib API.

10.3.2 The Object-Oriented API

Here's the code corresponding to the preceding figure using the object-oriented API

```
In [2]: fig, ax = plt.subplots()
        ax.plot(x, y, 'b-', linewidth=2)
        plt.show()
```

Here the call `fig, ax = plt.subplots()` returns a pair, where

- `fig` is a `Figure` instance—like a blank canvas.
- `ax` is an `AxesSubplot` instance—think of a frame for plotting in.

The `plot()` function is actually a method of `ax`.

While there's a bit more typing, the more explicit use of objects gives us better control.

This will become more clear as we go along.

10.3.3 Tweaks

Here we've changed the line to red and added a legend

```
In [3]: fig, ax = plt.subplots()
        ax.plot(x, y, 'r-', linewidth=2, label='sine function', alpha=0.6)
        ax.legend()
        plt.show()
```

We've also used `alpha` to make the line slightly transparent—which makes it look smoother.

The location of the legend can be changed by replacing `ax.legend()` with `ax.legend(loc='upper center')`.

```
In [4]: fig, ax = plt.subplots()
        ax.plot(x, y, 'r-', linewidth=2, label='sine function', alpha=0.6)
        ax.legend(loc='upper center')
        plt.show()
```

If everything is properly configured, then adding LaTeX is trivial

```
In [5]: fig, ax = plt.subplots()
        ax.plot(x, y, 'r-', linewidth=2, label='$y=\sin(x)$', alpha=0.6)
        ax.legend(loc='upper center')
        plt.show()
```

Controlling the ticks, adding titles and so on is also straightforward

```
In [6]: fig, ax = plt.subplots()
        ax.plot(x, y, 'r-', linewidth=2, label='$y=\sin(x)$', alpha=0.6)
        ax.legend(loc='upper center')
        ax.set_yticks([-1, 0, 1])
        ax.set_title('Test plot')
        plt.show()
```

10.4 More Features

Matplotlib has a huge array of functions and features, which you can discover over time as you have need for them.

We mention just a few.

10.4.1 Multiple Plots on One Axis

It's straightforward to generate multiple plots on the same axes.

Here's an example that randomly generates three normal densities and adds a label with their mean

```
In [7]: from scipy.stats import norm
        from random import uniform

        fig, ax = plt.subplots()
        x = np.linspace(-4, 4, 150)
        for i in range(3):
            m, s = uniform(-1, 1), uniform(1, 2)
            y = norm.pdf(x, loc=m, scale=s)
            current_label = f'$\mu = {m:.2}$'
            ax.plot(x, y, linewidth=2, alpha=0.6, label=current_label)
        ax.legend()
        plt.show()
```

10.4.2 Multiple Subplots

Sometimes we want multiple subplots in one figure.

Here's an example that generates 6 histograms

```
In [8]: num_rows, num_cols = 3, 2
        fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 12))
        for i in range(num_rows):
            for j in range(num_cols):
                m, s = uniform(-1, 1), uniform(1, 2)
                x = norm.rvs(loc=m, scale=s, size=100)
                axes[i, j].hist(x, alpha=0.6, bins=20)
                t = f'$\mu = {m:.2}, \quad \sigma = {s:.2}$'
                axes[i, j].set(title=t, xticks=[-4, 0, 4], yticks=[])
        plt.show()
```

10.4.3 3D Plots

Matplotlib does a nice job of 3D plots — here is one example

```
In [9]: from mpl_toolkits.mplot3d.axes3d import Axes3D
        from matplotlib import cm


        def f(x, y):
            return np.cos(x**2 + y**2) / (1 + x**2 + y**2)

        xgrid = np.linspace(-3, 3, 50)
```

```
ygrid = xgrid
x, y = np.meshgrid(xgrid, ygrid)

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x,
                y,
                f(x, y),
                rstride=2, cstride=2,
                cmap=cm.jet,
                alpha=0.7,
                linewidth=0.25)
ax.set_zlim(-0.5, 1.0)
plt.show()
```

10.4.4 A Customizing Function

Perhaps you will find a set of customizations that you regularly use.

Suppose we usually prefer our axes to go through the origin, and to have a grid.

Here's a nice example from Matthew Doty of how the object-oriented API can be used to build a custom `subplots` function that implements these changes.

Read carefully through the code and see if you can follow what's going on

```
In [10]: def subplots():
             "Custom subplots with axes through the origin"
```

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

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

    ax.grid()
    return fig, ax


fig, ax = subplots()  # Call the local version, not plt.subplots()
x = np.linspace(-2, 10, 200)
y = np.sin(x)
ax.plot(x, y, 'r-', linewidth=2, label='sine function', alpha=0.6)
ax.legend(loc='lower right')
plt.show()
```

The custom `subplots` function

1. calls the standard `plt.subplots` function internally to generate the `fig, ax` pair,
2. makes the desired customizations to `ax`, and
3. passes the `fig, ax` pair back to the calling code.

10.5 Further Reading

- The Matplotlib gallery provides many examples.
- A nice Matplotlib tutorial by Nicolas Rougier, Mike Muller and Gael Varoquaux.
- mpltools allows easy switching between plot styles.
- Seaborn facilitates common statistics plots in Matplotlib.

10.6 Exercises

10.6.1 Exercise 1

Plot the function

$$f(x) = \cos(\pi\theta x)\exp(-x)$$

over the interval $[0, 5]$ for each θ in `np.linspace(0, 2, 10)`.

Place all the curves in the same figure.

The output should look like this

10.7 Solutions

10.7.1 Exercise 1

Here's one solution

```
In [11]: def f(x, θ):
             return np.cos(np.pi * θ * x ) * np.exp(- x)

         θ_vals = np.linspace(0, 2, 10)
         x = np.linspace(0, 5, 200)
         fig, ax = plt.subplots()

         for θ in θ_vals:
             ax.plot(x, f(x, θ))

         plt.show()
```

1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0
1
2
3
4
5

Chapter 11

SciPy

11.1 Contents

- Overview 11.2
- SciPy versus NumPy 11.3
- Statistics 11.4
- Roots and Fixed Points 11.5
- Optimization 11.6
- Integration 11.7
- Linear Algebra 11.8
- Exercises 11.9
- Solutions 11.10

11.2 Overview

SciPy builds on top of NumPy to provide common tools for scientific programming such as

- linear algebra
- numerical integration
- interpolation
- optimization
- distributions and random number generation
- signal processing
- etc., etc

Like NumPy, SciPy is stable, mature and widely used.

Many SciPy routines are thin wrappers around industry-standard Fortran libraries such as LAPACK, BLAS, etc.

It's not really necessary to "learn" SciPy as a whole.

A more common approach is to get some idea of what's in the library and then look up documentation as required.

In this lecture, we aim only to highlight some useful parts of the package.

11.3 SciPy versus NumPy

SciPy is a package that contains various tools that are built on top of NumPy, using its array data type and related functionality.

In fact, when we import SciPy we also get NumPy, as can be seen from this excerpt the SciPy initialization file:

```
In [1]: # Import numpy symbols to scipy namespace
        from numpy import *
        from numpy.random import rand, randn
        from numpy.fft import fft, ifft
        from numpy.lib.scimath import *
```

However, it's more common and better practice to use NumPy functionality explicitly

```
In [2]: import numpy as np

        a = np.identity(3)
```

What is useful in SciPy is the functionality in its sub-packages

- `scipy.optimize`, `scipy.integrate`, `scipy.stats`, etc.

Let's explore some of the major sub-packages.

11.4 Statistics

The `scipy.stats` subpackage supplies

- numerous random variable objects (densities, cumulative distributions, random sampling, etc.)
- some estimation procedures
- some statistical tests

11.4.1 Random Variables and Distributions

Recall that `numpy.random` provides functions for generating random variables

```
In [3]: np.random.beta(5, 5, size=3)

Out[3]: array([0.38169482, 0.45705347, 0.41404047])
```

This generates a draw from the distribution with the density function below when `a, b = 5, 5`

$$f(x; a, b) = \frac{x^{(a-1)}(1-x)^{(b-1)}}{\int_0^1 u^{(a-1)}(1-u)^{(b-1)}du} \qquad (0 \leq x \leq 1) \tag{1}$$

Sometimes we need access to the density itself, or the cdf, the quantiles, etc.

For this, we can use `scipy.stats`, which provides all of this functionality as well as random number generation in a single consistent interface.

Here's an example of usage

```
In [4]: from scipy.stats import beta
        import matplotlib.pyplot as plt
        %matplotlib inline

        q = beta(5, 5)      # Beta(a, b), with a = b = 5
        obs = q.rvs(2000)   # 2000 observations
        grid = np.linspace(0.01, 0.99, 100)

        fig, ax = plt.subplots()
        ax.hist(obs, bins=40, density=True)
        ax.plot(grid, q.pdf(grid), 'k-', linewidth=2)
        plt.show()
```

The object `q` that represents the distribution has additional useful methods, including

```
In [5]: q.cdf(0.4)      # Cumulative distribution function

Out[5]: 0.26656768000000003

In [6]: q.ppf(0.8)      # Quantile (inverse cdf) function

Out[6]: 0.6339134834642708

In [7]: q.mean()

Out[7]: 0.5
```

The general syntax for creating these objects that represent distributions (of type `rv_frozen`) is

```
name = scipy.stats.distribution_name(shape_parameters,
loc=c, scale=d)
```

Here `distribution_name` is one of the distribution names in scipy.stats.

The `loc` and `scale` parameters transform the original random variable X into $Y = c + dX$.

11.4.2 Alternative Syntax

There is an alternative way of calling the methods described above.

For example, the code that generates the figure above can be replaced by

```
In [8]: obs = beta.rvs(5, 5, size=2000)
        grid = np.linspace(0.01, 0.99, 100)

        fig, ax = plt.subplots()
        ax.hist(obs, bins=40, density=True)
        ax.plot(grid, beta.pdf(grid, 5, 5), 'k-', linewidth=2)
        plt.show()
```

11.4.3 Other Goodies in scipy.stats

There are a variety of statistical functions in `scipy.stats`.

For example, `scipy.stats.linregress` implements simple linear regression

```
In [9]: from scipy.stats import linregress

        x = np.random.randn(200)
        y = 2 * x + 0.1 * np.random.randn(200)
        gradient, intercept, r_value, p_value, std_err = linregress(x, y)
        gradient, intercept
```

```
Out[9]: (2.001780548290325, 0.000979142986114484)
```

To see the full list, consult the documentation.

11.5 Roots and Fixed Points

A **root** or **zero** of a real function f on $[a, b]$ is an $x \in [a, b]$ such that $f(x) = 0$.

For example, if we plot the function

$$f(x) = \sin(4(x - 1/4)) + x + x^{20} - 1 \tag{2}$$

with $x \in [0, 1]$ we get

```
In [10]: f = lambda x: np.sin(4 * (x - 1/4)) + x + x**20 - 1
         x = np.linspace(0, 1, 100)

         fig, ax = plt.subplots()
         ax.plot(x, f(x))
         ax.axhline(ls='--', c='k', label='$f(x)$')
         ax.set_xlabel('$x$', fontsize=12)
         ax.set_ylabel('$f(x)$', fontsize=12)
         ax.legend(fontsize=12)
         plt.show()
```

The unique root is approximately 0.408.

Let's consider some numerical techniques for finding roots.

11.5.1 Bisection

One of the most common algorithms for numerical root-finding is *bisection*.

To understand the idea, recall the well-known game where

- Player A thinks of a secret number between 1 and 100
- Player B asks if it's less than 50
  - If yes, B asks if it's less than 25
  - If no, B asks if it's less than 75

And so on.

This is bisection.

Here's a simplistic implementation of the algorithm in Python.

It works for all sufficiently well behaved increasing continuous functions with $f(a) < 0 < f(b)$

```
In [11]: def bisect(f, a, b, tol=10e-5):
             """
             Implements the bisection root finding algorithm, assuming that f is a
             real-valued function on [a, b] satisfying f(a) < 0 < f(b).
             """
             lower, upper = a, b

             while upper - lower > tol:
                 middle = 0.5 * (upper + lower)
                 if f(middle) > 0:   # root is between lower and middle
                     lower, upper = lower, middle
                 else:               # root is between middle and upper
                     lower, upper = middle, upper

             return 0.5 * (upper + lower)
```

Let's test it using the function f defined in (2)

```
In [12]: bisect(f, 0, 1)

Out[12]: 0.408294677734375
```

Not surprisingly, SciPy provides its own bisection function.

Let's test it using the same function f defined in (2)

```
In [13]: from scipy.optimize import bisect

         bisect(f, 0, 1)

Out[13]: 0.4082935042806639
```

11.5.2 The Newton-Raphson Method

Another very common root-finding algorithm is the Newton-Raphson method.

In SciPy this algorithm is implemented by `scipy.optimize.newton`.

Unlike bisection, the Newton-Raphson method uses local slope information in an attempt to increase the speed of convergence.

Let's investigate this using the same function f defined above.

With a suitable initial condition for the search we get convergence:

```
In [14]: from scipy.optimize import newton

         newton(f, 0.2)   # Start the search at initial condition x = 0.2

Out[14]: 0.40829350427935673
```

But other initial conditions lead to failure of convergence:

```
In [15]: newton(f, 0.7)   # Start the search at x = 0.7 instead

Out[15]: 0.7001700000000279
```

11.5.3 Hybrid Methods

A general principle of numerical methods is as follows:

- If you have specific knowledge about a given problem, you might be able to exploit it to generate efficiency.
- If not, then the choice of algorithm involves a trade-off between speed and robustness.

In practice, most default algorithms for root-finding, optimization and fixed points use *hybrid* methods.

These methods typically combine a fast method with a robust method in the following manner:

1. Attempt to use a fast method
2. Check diagnostics
3. If diagnostics are bad, then switch to a more robust algorithm

In `scipy.optimize`, the function `brentq` is such a hybrid method and a good default

```
In [16]: from scipy.optimize import brentq

         brentq(f, 0, 1)

Out[16]: 0.40829350427936706
```

Here the correct solution is found and the speed is better than bisection:

In [17]:
```
%timeit brentq(f, 0, 1)
```

```
28.5 µs ± 862 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

In [18]:
```
%timeit bisect(f, 0, 1)
```

```
109 µs ± 1.91 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

11.5.4 Multivariate Root-Finding

Use `scipy.optimize.fsolve`, a wrapper for a hybrid method in MINPACK.

See the documentation for details.

11.5.5 Fixed Points

A **fixed point** of a real function f on $[a, b]$ is an $x \in [a, b]$ such that $f(x) = x$.

SciPy has a function for finding (scalar) fixed points too

In [19]:
```
from scipy.optimize import fixed_point

fixed_point(lambda x: x**2, 10.0)  # 10.0 is an initial guess
```

Out[19]:
```
array(1.)
```

If you don't get good results, you can always switch back to the `brentq` root finder, since the fixed point of a function f is the root of $g(x) := x - f(x)$.

11.6 Optimization

Most numerical packages provide only functions for *minimization*.

Maximization can be performed by recalling that the maximizer of a function f on domain D is the minimizer of $-f$ on D.

Minimization is closely related to root-finding: For smooth functions, interior optima correspond to roots of the first derivative.

The speed/robustness trade-off described above is present with numerical optimization too.

Unless you have some prior information you can exploit, it's usually best to use hybrid methods.

For constrained, univariate (i.e., scalar) minimization, a good hybrid option is `fminbound`

In [20]:
```
from scipy.optimize import fminbound

fminbound(lambda x: x**2, -1, 2)  # Search in [-1, 2]
```

Out[20]:
```
0.0
```

11.6.1 Multivariate Optimization

Multivariate local optimizers include `minimize`, `fmin`, `fmin_powell`, `fmin_cg`, `fmin_bfgs`, and `fmin_ncg`.

Constrained multivariate local optimizers include `fmin_l_bfgs_b`, `fmin_tnc`, `fmin_cobyla`.

See the documentation for details.

11.7 Integration

Most numerical integration methods work by computing the integral of an approximating polynomial.

The resulting error depends on how well the polynomial fits the integrand, which in turn depends on how "regular" the integrand is.

In SciPy, the relevant module for numerical integration is `scipy.integrate`.

A good default for univariate integration is `quad`

In [21]:
```
from scipy.integrate import quad

integral, error = quad(lambda x: x**2, 0, 1)
integral
```

Out[21]:
```
0.33333333333333337
```

In fact, `quad` is an interface to a very standard numerical integration routine in the Fortran library QUADPACK.

It uses Clenshaw-Curtis quadrature, based on expansion in terms of Chebychev polynomials.

There are other options for univariate integration—a useful one is `fixed_quad`, which is fast and hence works well inside `for` loops.

There are also functions for multivariate integration.

See the documentation for more details.

11.8 Linear Algebra

We saw that NumPy provides a module for linear algebra called `linalg`.

SciPy also provides a module for linear algebra with the same name.

The latter is not an exact superset of the former, but overall it has more functionality.

We leave you to investigate the set of available routines.

11.9 Exercises

11.9.1 Exercise 1

Previously we discussed the concept of recursive function calls.

Try to write a recursive implementation of homemade bisection function described above.

Test it on the function (2).

11.10 Solutions

11.10.1 Exercise 1

Here's a reasonable solution:

```
In [22]: def bisect(f, a, b, tol=10e-5):
             """
             Implements the bisection root-finding algorithm, assuming that f is a
             real-valued function on [a, b] satisfying f(a) < 0 < f(b).
             """
             lower, upper = a, b
             if upper - lower < tol:
                 return 0.5 * (upper + lower)
             else:
                 middle = 0.5 * (upper + lower)
                 print(f'Current mid point = {middle}')
                 if f(middle) > 0:   # Implies root is between lower and middle
                     return bisect(f, lower, middle)
                 else:               # Implies root is between middle and upper
                     return bisect(f, middle, upper)
```

We can test it as follows

```
In [23]: f = lambda x: np.sin(4 * (x - 0.25)) + x + x**20 - 1
         bisect(f, 0, 1)
```

```
Current mid point = 0.5
Current mid point = 0.25
Current mid point = 0.375
Current mid point = 0.4375
Current mid point = 0.40625
Current mid point = 0.421875
Current mid point = 0.4140625
Current mid point = 0.41015625
Current mid point = 0.408203125
Current mid point = 0.4091796875
Current mid point = 0.40869140625
Current mid point = 0.408447265625
Current mid point = 0.4083251953125
Current mid point = 0.40826416015625
```

```
Out[23]: 0.408294677734375
```

Chapter 12

Numba

12.1 Contents

- Overview 12.2
- Compiling Functions 12.3
- Decorators and "nopython" Mode **??**
- Compiling Classes 12.5
- Alternatives to Numba 12.6
- Summary and Comments 12.7
- Exercises 12.8
- Solutions 12.9

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

```
In [1]: !pip install --upgrade quantecon
```

Please also make sure that you have the latest version of Anaconda, since old versions are a common source of errors.

Let's start with some imports:

```
In [2]: import numpy as np
        import quantecon as qe
        import matplotlib.pyplot as plt

        %matplotlib inline
```

12.2 Overview

In an earlier lecture we learned about vectorization, which is one method to improve speed and efficiency in numerical work.

Vectorization involves sending array processing operations in batch to efficient low-level code.

However, as discussed previously, vectorization has several weaknesses.

One is that it is highly memory-intensive when working with large amounts of data.

Another is that the set of algorithms that can be entirely vectorized is not universal.

In fact, for some algorithms, vectorization is ineffective.

Fortunately, a new Python library called Numba solves many of these problems.

It does so through something called **just in time (JIT) compilation**.

The key idea is to compile functions to native machine code instructions on the fly.

When it succeeds, the compiled code is extremely fast.

Numba is specifically designed for numerical work and can also do other tricks such as multi-threading.

Numba will be a key part of our lectures — especially those lectures involving dynamic programming.

This lecture introduces the main ideas.

12.3 Compiling Functions

As stated above, Numba's primary use is compiling functions to fast native machine code during runtime.

12.3.1 An Example

Let's consider a problem that is difficult to vectorize: generating the trajectory of a difference equation given an initial condition.

We will take the difference equation to be the quadratic map

$$x_{t+1} = \alpha x_t (1 - x_t)$$

In what follows we set

```
In [3]: α = 4.0
```

Here's the plot of a typical trajectory, starting from $x_0 = 0.1$, with t on the x-axis

```
In [4]: def qm(x0, n):
            x = np.empty(n+1)
            x[0] = x0
            for t in range(n):
              x[t+1] = α * x[t] * (1 - x[t])
            return x

        x = qm(0.1, 250)
        fig, ax = plt.subplots()
        ax.plot(x, 'b-', lw=2, alpha=0.8)
        ax.set_xlabel('$t$', fontsize=12)
        ax.set_ylabel('$x_{t}$', fontsize = 12)
        plt.show()
```

To speed the function `qm` up using Numba, our first step is

```
In [5]: from numba import jit

        qm_numba = jit(qm)
```

The function `qm_numba` is a version of `qm` that is "targeted" for JIT-compilation.

We will explain what this means momentarily.

Let's time and compare identical function calls across these two versions, starting with the original function `qm`:

```
In [6]: n = 10_000_000

        qe.tic()
        qm(0.1, int(n))
        time1 = qe.toc()
```

```
TOC: Elapsed: 0:00:10.41
```

Now let's try qm_numba

```
In [7]: qe.tic()
        qm_numba(0.1, int(n))
        time2 = qe.toc()
```

```
TOC: Elapsed: 0:00:0.19
```

This is already a massive speed gain.

In fact, the next time and all subsequent times it runs even faster as the function has been compiled and is in memory:

```
In [8]: qe.tic()
        qm_numba(0.1, int(n))
        time3 = qe.toc()
```

```
TOC: Elapsed: 0:00:0.07
```

```
In [9]: time1 / time3  # Calculate speed gain
```

```
Out[9]: 137.20886417891694
```

This kind of speed gain is huge relative to how simple and clear the implementation is.

12.3.2 How and When it Works

Numba attempts to generate fast machine code using the infrastructure provided by the LLVM Project.

It does this by inferring type information on the fly.

(See our earlier lecture on scientific computing for a discussion of types.)

The basic idea is this:

- Python is very flexible and hence we could call the function qm with many types.
  - e.g., `x0` could be a NumPy array or a list, `n` could be an integer or a float, etc.
- This makes it hard to *pre*-compile the function.
- However, when we do actually call the function, say by executing `qm(0.5, 10)`, the types of `x0` and `n` become clear.
- Moreover, the types of other variables in `qm` can be inferred once the input is known.
- So the strategy of Numba and other JIT compilers is to wait until this moment, and *then* compile the function.

That's why it is called "just-in-time" compilation.

Note that, if you make the call `qm(0.5, 10)` and then follow it with `qm(0.9, 20)`, compilation only takes place on the first call.

The compiled code is then cached and recycled as required.

12.4 Decorators and "nopython" Mode

In the code above we created a JIT compiled version of `qm` via the call

```
In [10]: qm_numba = jit(qm)
```

In practice this would typically be done using an alternative *decorator* syntax.

(We will explain all about decorators in a later lecture but you can skip the details at this stage.)

Let's see how this is done.

12.4.1 Decorator Notation

To target a function for JIT compilation we can put `@jit` before the function definition.

Here's what this looks like for `qm`

```
In [11]: @jit
         def qm(x0, n):
             x = np.empty(n+1)
             x[0] = x0
             for t in range(n):
                 x[t+1] = α * x[t] * (1 - x[t])
             return x
```

This is equivalent to `qm = jit(qm)`.

The following now uses the jitted version:

```
In [12]: qm(0.1, 10)
```

```
Out[12]: array([0.1       , 0.36      , 0.9216    , 0.28901376, 0.82193923,
                0.58542054, 0.97081333, 0.11333925, 0.40197385, 0.9615635 ,
                0.14783656])
```

12.4.2 Type Inference and "nopython" Mode

Clearly type inference is a key part of JIT compilation.

As you can imagine, inferring types is easier for simple Python objects (e.g., simple scalar data types such as floats and integers).

Numba also plays well with NumPy arrays.

In an ideal setting, Numba can infer all necessary type information.

This allows it to generate native machine code, without having to call the Python runtime environment.

In such a setting, Numba will be on par with machine code from low-level languages.

When Numba cannot infer all type information, some Python objects are given generic object status and execution falls back to the Python runtime.

When this happens, Numba provides only minor speed gains or none at all.

We generally prefer to force an error when this occurs, so we know effective compilation is failing.

This is done by using either `@jit(nopython=True)` or, equivalently, `@njit` instead of `@jit`.

For example,

In [13]:
```
from numba import njit

@njit
def qm(x0, n):
    x = np.empty(n+1)
    x[0] = x0
    for t in range(n):
        x[t+1] = 4 * x[t] * (1 - x[t])
    return x
```

12.5 Compiling Classes

As mentioned above, at present Numba can only compile a subset of Python.

However, that subset is ever expanding.

For example, Numba is now quite effective at compiling classes.

If a class is successfully compiled, then its methods act as JIT-compiled functions.

To give one example, let's consider the class for analyzing the Solow growth model we created in this lecture.

To compile this class we use the `@jitclass` decorator:

In [14]:
```
from numba import jitclass, float64
```

Notice that we also imported something called `float64`.

This is a data type representing standard floating point numbers.

We are importing it here because Numba needs a bit of extra help with types when it trys to deal with classes.

Here's our code:

In [15]:
```
solow_data = [
    ('n', float64),
    ('s', float64),
    ('δ', float64),
    ('α', float64),
    ('z', float64),
    ('k', float64)
]

@jitclass(solow_data)
class Solow:
    r"""
    Implements the Solow growth model with the update rule

        k_{t+1} = [(s z k^α_t) + (1 - δ)k_t] /(1 + n)

    """
    def __init__(self, n=0.05,  # population growth rate
                       s=0.25,  # savings rate
                       δ=0.1,   # depreciation rate
                       α=0.3,   # share of labor
```

```
                        z=2.0,    # productivity
                        k=1.0):   # current capital stock

        self.n, self.s, self.δ, self.α, self.z = n, s, δ, α, z
        self.k = k

    def h(self):
        "Evaluate the h function"
        # Unpack parameters (get rid of self to simplify notation)
        n, s, δ, α, z = self.n, self.s, self.δ, self.α, self.z
        # Apply the update rule
        return (s * z * self.k**α + (1 - δ) * self.k) / (1 + n)

    def update(self):
        "Update the current state (i.e., the capital stock)."
        self.k =  self.h()

    def steady_state(self):
        "Compute the steady state value of capital."
        # Unpack parameters (get rid of self to simplify notation)
        n, s, δ, α, z = self.n, self.s, self.δ, self.α, self.z
        # Compute and return steady state
        return ((s * z) / (n + δ))**(1 / (1 - α))

    def generate_sequence(self, t):
        "Generate and return a time series of length t"
        path = []
        for i in range(t):
            path.append(self.k)
            self.update()
        return path
```

First we specified the types of the instance data for the class in `solow_data`.

After that, targeting the class for JIT compilation only requires adding `@jitclass(solow_data)` before the class definition.

When we call the methods in the class, the methods are compiled just like functions.

```
In [16]: s1 = Solow()
         s2 = Solow(k=8.0)

         T = 60
         fig, ax = plt.subplots()

         # Plot the common steady state value of capital
         ax.plot([s1.steady_state()]*T, 'k-', label='steady state')

         # Plot time series for each economy
         for s in s1, s2:
             lb = f'capital series from initial state {s.k}'
             ax.plot(s.generate_sequence(T), 'o-', lw=2, alpha=0.6, label=lb)
         ax.set_ylabel('$k_{t}$', fontsize=12)
         ax.set_xlabel('$t$', fontsize=12)
         ax.legend()
         plt.show()
```

12.6 Alternatives to Numba

There are additional options for accelerating Python loops.

Here we quickly review them.

However, we do so only for interest and completeness.

If you prefer, you can safely skip this section.

12.6.1 Cython

Like Numba, Cython provides an approach to generating fast compiled code that can be used from Python.

As was the case with Numba, a key problem is the fact that Python is dynamically typed.

As you'll recall, Numba solves this problem (where possible) by inferring type.

Cython's approach is different — programmers add type definitions directly to their "Python" code.

As such, the Cython language can be thought of as Python with type definitions.

In addition to a language specification, Cython is also a language translator, transforming Cython code into optimized C and C++ code.

Cython also takes care of building language extensions — the wrapper code that interfaces between the resulting compiled code and Python.

While Cython has certain advantages, we generally find it both slower and more cumbersome than Numba.

12.6.2 Interfacing with Fortran via F2Py

If you are comfortable writing Fortran you will find it very easy to create extension modules from Fortran code using F2Py.

F2Py is a Fortran-to-Python interface generator that is particularly simple to use.

Robert Johansson provides a nice introduction to F2Py, among other things.

Recently, a Jupyter cell magic for Fortran has been developed — you might want to give it a try.

12.7 Summary and Comments

Let's review the above and add some cautionary notes.

12.7.1 Limitations

As we've seen, Numba needs to infer type information on all variables to generate fast machine-level instructions.

For simple routines, Numba infers types very well.

For larger ones, or for routines using external libraries, it can easily fail.

Hence, it's prudent when using Numba to focus on speeding up small, time-critical snippets of code.

This will give you much better performance than blanketing your Python programs with `@jit` statements.

12.7.2 A Gotcha: Global Variables

Here's another thing to be careful about when using Numba.

Consider the following example

In [17]:
```
a = 1

@jit
def add_a(x):
    return a + x

print(add_a(10))
```

```
11
```

In [18]:
```
a = 2

print(add_a(10))
```

```
11
```

Notice that changing the global had no effect on the value returned by the function.

When Numba compiles machine code for functions, it treats global variables as constants to ensure type stability.

12.8 Exercises

12.8.1 Exercise 1

Previously we considered how to approximate π by Monte Carlo.

Use the same idea here, but make the code efficient using Numba.

Compare speed with and without Numba when the sample size is large.

12.8.2 Exercise 2

Later we'll learn all about finite-state Markov chains.

For now, let's just concentrate on simulating a very simple example of such a chain.

Suppose that the volatility of returns on an asset can be in one of two regimes — high or low.

The transition probabilities across states are as follows

For example, let the period length be one day, and suppose the current state is high.

We see from the graph that the state tomorrow will be

- high with probability 0.8
- low with probability 0.2

Your task is to simulate a sequence of daily volatility states according to this rule.

Set the length of the sequence to `n = 1_000_000` and start in the high state.

Implement a pure Python version and a Numba version, and compare speeds.

To test your code, evaluate the fraction of time that the chain spends in the low state.

If your code is correct, it should be about 2/3.

Hints:

- Represent the low state as 0 and the high state as 1.
- If you want to store integers in a NumPy array and then apply JIT compilation, use `x = np.empty(n, dtype=np.int_)`.

12.9 Solutions

12.9.1 Exercise 1

Here is one solution:

In [19]:
```
from random import uniform

@njit
def calculate_pi(n=1_000_000):
    count = 0
    for i in range(n):
        u, v = uniform(0, 1), uniform(0, 1)
        d = np.sqrt((u - 0.5)**2 + (v - 0.5)**2)
        if d < 0.5:
            count += 1

    area_estimate = count / n
    return area_estimate * 4  # dividing by radius**2
```

Now let's see how fast it runs:

In [20]:
```
%time calculate_pi()
```

```
    CPU times: user 234 ms, sys: 111 µs, total: 235 ms
Wall time: 245 ms
```

Out[20]:
```
3.141832
```

In [21]:
```
%time calculate_pi()
```

```
    CPU times: user 25.5 ms, sys: 0 ns, total: 25.5 ms
Wall time: 30.6 ms
```

Out[21]:
```
3.142736
```

If we switch of JIT compilation by removing `@njit`, the code takes around 150 times as long on our machine.

So we get a speed gain of 2 orders of magnitude–which is huge–by adding four characters.

12.9.2 Exercise 2

We let

- 0 represent "low"
- 1 represent "high"

In [22]:
```
p, q = 0.1, 0.2  # Prob of leaving low and high state respectively
```

Here's a pure Python version of the function

```
In [23]: def compute_series(n):
             x = np.empty(n, dtype=np.int_)
             x[0] = 1  # Start in state 1
             U = np.random.uniform(0, 1, size=n)
             for t in range(1, n):
                 current_x = x[t-1]
                 if current_x == 0:
                     x[t] = U[t] < p
                 else:
                     x[t] = U[t] > q
             return x
```

Let's run this code and check that the fraction of time spent in the low state is about 0.666

```
In [24]: n = 1_000_000
         x = compute_series(n)
         print(np.mean(x == 0))  # Fraction of time x is in state 0
```

```
0.668291
```

This is (approximately) the right output.

Now let's time it:

```
In [25]: qe.tic()
         compute_series(n)
         qe.toc()
```

```
TOC: Elapsed: 0:00:1.28
```

```
Out[25]: 1.2830963134765625
```

Next let's implement a Numba version, which is easy

```
In [26]: from numba import jit

         compute_series_numba = jit(compute_series)
```

Let's check we still get the right numbers

```
In [27]: x = compute_series_numba(n)
         print(np.mean(x == 0))
```

```
0.667422
```

Let's see the time

In [28]:
```
qe.tic()
compute_series_numba(n)
qe.toc()
```

```
TOC: Elapsed: 0:00:0.01
```

Out[28]:
```
0.019926786422729492
```

This is a nice speed improvement for one line of code!

Chapter 13

Parallelization

13.1 Contents

- Overview 13.2
- Types of Parallelization 13.3
- Implicit Multithreading in NumPy 13.4
- Multithreaded Loops in Numba 13.5
- Exercises 13.6
- Solutions 13.7

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

```
In [1]: !pip install --upgrade quantecon
```

Co-author: Matthew McKay

13.2 Overview

The growth of CPU clock speed (i.e., the speed at which a single chain of logic can be run) has slowed dramatically in recent years.

This is unlikely to change in the near future, due to inherent physical limitations on the construction of chips and circuit boards.

Chip designers and computer programmers have responded to the slowdown by seeking a different path to fast execution: parallelization.

Hardware makers have increased the number of cores (physical CPUs) embedded in each machine.

For programmers, the challenge has been to exploit these multiple CPUs by running many processes in parallel (i.e., simultaneously).

This is particularly important in scientific programming, which requires handling

- large amounts of data and
- CPU intensive simulations and other calculations.

In this lecture we discuss parallelization for scientific computing, with a focus on

1. the best tools for parallelization in Python and
2. how these tools can be applied to quantitative economic problems.

Let's start with some imports:

```
In [2]: import numpy as np
        import quantecon as qe
        import matplotlib.pyplot as plt

        %matplotlib inline
```

13.3 Types of Parallelization

Large textbooks have been written on different approaches to parallelization but we will keep a tight focus on what's most useful to us.

We will briefly review the two main kinds of parallelization commonly used in scientific computing and discuss their pros and cons.

13.3.1 Multiprocessing

Multiprocessing means concurrent execution of multiple processes using more than one processor.

In this context, a **process** is a chain of instructions (i.e., a program).

Multiprocessing can be carried out on one machine with multiple CPUs or on a collection of machines connected by a network.

In the latter case, the collection of machines is usually called a **cluster**.

With multiprocessing, each process has its own memory space, although the physical memory chip might be shared.

13.3.2 Multithreading

Multithreading is similar to multiprocessing, except that, during execution, the threads all share the same memory space.

Native Python struggles to implement multithreading due to some legacy design features.

But this is not a restriction for scientific libraries like NumPy and Numba.

Functions imported from these libraries and JIT-compiled code run in low level execution environments where Python's legacy restrictions don't apply.

13.3.3 Advantages and Disadvantages

Multithreading is more lightweight because most system and memory resources are shared by the threads.

In addition, the fact that multiple threads all access a shared pool of memory is extremely convenient for numerical programming.

On the other hand, multiprocessing is more flexible and can be distributed across clusters.

For the great majority of what we do in these lectures, multithreading will suffice.

13.4 Implicit Multithreading in NumPy

Actually, you have already been using multithreading in your Python code, although you might not have realized it.

(We are, as usual, assuming that you are running the latest version of Anaconda Python.)

This is because NumPy cleverly implements multithreading in a lot of its compiled code.

Let's look at some examples to see this in action.

13.4.1 A Matrix Operation

The next piece of code computes the eigenvalues of a large number of randomly generated matrices.

It takes a few seconds to run.

```
In [3]: n = 20
        m = 1000
        for i in range(n):
            X = np.random.randn(m, m)
            λ = np.linalg.eigvals(X)
```

Now, let's look at the output of the htop system monitor on our machine while this code is running:

We can see that 4 of the 8 CPUs are running at full speed.

This is because NumPy's `eigvals` routine neatly splits up the tasks and distributes them to different threads.

13.4.2 A Multithreaded Ufunc

Over the last few years, NumPy has managed to push this kind of multithreading out to more and more operations.

For example, let's return to a maximization problem discussed previously:

```
In [4]: def f(x, y):
            return np.cos(x**2 + y**2) / (1 + x**2 + y**2)

        grid = np.linspace(-3, 3, 5000)
        x, y = np.meshgrid(grid, grid)
```

```
In [5]: %timeit np.max(f(x, y))
```

```
863 ms ± 23.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

If you have a system monitor such as htop (Linux/Mac) or perfmon (Windows), then try running this and then observing the load on your CPUs.

(You will probably need to bump up the grid size to see large effects.)

At least on our machine, the output shows that the operation is successfully distributed across multiple threads.

This is one of the reasons why the vectorized code above is fast.

13.4.3 A Comparison with Numba

To get some basis for comparison for the last example, let's try the same thing with Numba.

In fact there is an easy way to do this, since Numba can also be used to create custom ufuncs with the [@vectorize](http://numba.pydata.org/numba-doc/dev/user/vectorize.html) decorator.

```
In [6]: from numba import vectorize

        @vectorize
        def f_vec(x, y):
            return np.cos(x**2 + y**2) / (1 + x**2 + y**2)

        np.max(f_vec(x, y))  # Run once to compile
```

```
Out[6]: 0.9999992797121728
```

```
In [7]: %timeit np.max(f_vec(x, y))
```

```
587 ms ± 3.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

At least on our machine, the difference in the speed between the Numba version and the vectorized NumPy version shown above is not large.

But there's quite a bit going on here so let's try to break down what is happening.

Both Numba and NumPy use efficient machine code that's specialized to these floating point operations.

However, the code NumPy uses is, in some ways, less efficient.

The reason is that, in NumPy, the operation `np.cos(x**2 + y**2) / (1 + x**2 + y**2)` generates several intermediate arrays.

For example, a new array is created when `x**2` is calculated.

The same is true when `y**2` is calculated, and then `x**2 + y**2` and so on.

Numba avoids creating all these intermediate arrays by compiling one function that is specialized to the entire operation.

But if this is true, then why isn't the Numba code faster?

The reason is that NumPy makes up for its disadvantages with implicit multithreading, as we've just discussed.

13.4.4 Multithreading a Numba Ufunc

Can we get both of these advantages at once?

In other words, can we pair

- the efficiency of Numba's highly specialized JIT compiled function and
- the speed gains from parallelization obtained by NumPy's implicit multithreading?

It turns out that we can, by adding some type information plus `target='parallel'`.

```
In [8]: @vectorize('float64(float64, float64)', target='parallel')
        def f_vec(x, y):
            return np.cos(x**2 + y**2) / (1 + x**2 + y**2)

        np.max(f_vec(x, y))  # Run once to compile
```

```
Out[8]: 0.9999992797121728
```

```
In [9]: %timeit np.max(f_vec(x, y))
```

```
476 ms ± 2.53 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Now our code runs significantly faster than the NumPy version.

13.5 Multithreaded Loops in Numba

We just saw one approach to parallelization in Numba, using the `parallel` flag in `@vectorize`.

This is neat but, it turns out, not well suited to many problems we consider.

Fortunately, Numba provides another approach to multithreading that will work for us almost everywhere parallelization is possible.

To illustrate, let's look first at a simple, single-threaded (i.e., non-parallelized) piece of code.

The code simulates updating the wealth w_t of a household via the rule

$$w_{t+1} = R_{t+1} s w_t + y_{t+1}$$

Here

- R is the gross rate of return on assets
- s is the savings rate of the household and
- y is labor income.

We model both R and y as independent draws from a lognormal distribution.

Here's the code:

In [10]:
```
from numpy.random import randn
from numba import njit

@njit
def h(w, r=0.1, s=0.3, v1=0.1, v2=1.0):
    """
    Updates household wealth.
    """

    # Draw shocks
    R = np.exp(v1 * randn()) * (1 + r)
    y = np.exp(v2 * randn())

    # Update wealth
    w = R * s * w + y
    return w
```

Let's have a look at how wealth evolves under this rule.

In [11]:
```
fig, ax = plt.subplots()

T = 100
w = np.empty(T)
w[0] = 5
for t in range(T-1):
    w[t+1] = h(w[t])

ax.plot(w)
ax.set_xlabel('$t$', fontsize=12)
ax.set_ylabel('$w_{t}$', fontsize=12)
plt.show()
```

Now let's suppose that we have a large population of households and we want to know what median wealth will be.

This is not easy to solve with pencil and paper, so we will use simulation instead.

In particular, we will simulate a large number of households and then calculate median wealth for this group.

Suppose we are interested in the long-run average of this median over time.

It turns out that, for the specification that we've chosen above, we can calculate this by taking a one-period snapshot of what has happened to median wealth of the group at the end of a long simulation.

Moreover, provided the simulation period is long enough, initial conditions don't matter.

- This is due to something called ergodicity, which we will discuss later on.

So, in summary, we are going to simulate 50,000 households by

1. arbitrarily setting initial wealth to 1 and
2. simulating forward in time for 1,000 periods.

Then we'll calculate median wealth at the end period.

Here's the code:

```
In [12]: @njit
         def compute_long_run_median(w0=1, T=1000, num_reps=50_000):

             obs = np.empty(num_reps)
             for i in range(num_reps):
                 w = w0
```

```
        for t in range(T):
            w = h(w)
        obs[i] = w

    return np.median(obs)
```

Let's see how fast this runs:

```
In [13]: %%time
         compute_long_run_median()
```

```
    CPU times: user 8.95 s, sys: 16.4 ms, total: 8.96 s
Wall time: 8.95 s
```

```
Out[13]: 1.8461123440407574
```

To speed this up, we're going to parallelize it via multithreading.

To do so, we add the `parallel=True` flag and change `range` to `prange`:

```
In [14]: from numba import prange

         @njit(parallel=True)
         def compute_long_run_median_parallel(w0=1, T=1000, num_reps=50_000):

             obs = np.empty(num_reps)
             for i in prange(num_reps):
                 w = w0
                 for t in range(T):
                     w = h(w)
                 obs[i] = w

             return np.median(obs)
```

Let's look at the timing:

```
In [15]: %%time
         compute_long_run_median_parallel()
```

```
    CPU times: user 9.77 s, sys: 7.93 ms, total: 9.77 s
Wall time: 5.54 s
```

```
Out[15]: 1.837654164804717
```

The speed-up is significant.

13.5.1 A Warning

Parallelization works well in the outer loop of the last example because the individual tasks inside the loop are independent of each other.

If this independence fails then parallelization is often problematic.

For example, each step inside the inner loop depends on the last step, so independence fails, and this is why we use ordinary `range` instead of `prange`.

When you see us using `prange` in later lectures, it is because the independence of tasks holds true.

When you see us using ordinary `range` in a jitted function, it is either because the speed gain from parallelization is small or because independence fails.

13.6 Exercises

13.6.1 Exercise 1

In an earlier exercise, we used Numba to accelerate an effort to compute the constant π by Monte Carlo.

Now try adding parallelization and see if you get further speed gains.

You should not expect huge gains here because, while there are many independent tasks (draw point and test if in circle), each one has low execution time.

Generally speaking, parallelization is less effective when the individual tasks to be parallelized are very small relative to total execution time.

This is due to overheads associated with spreading all of these small tasks across multiple CPUs.

Nevertheless, with suitable hardware, it is possible to get nontrivial speed gains in this exercise.

For the size of the Monte Carlo simulation, use something substantial, such as `n = 100_000_000`.

13.7 Solutions

13.7.1 Exercise 1

Here is one solution:

In [16]:
```
from random import uniform

@njit(parallel=True)
def calculate_pi(n=1_000_000):
    count = 0
    for i in prange(n):
        u, v = uniform(0, 1), uniform(0, 1)
        d = np.sqrt((u - 0.5)**2 + (v - 0.5)**2)
        if d < 0.5:
            count += 1

    area_estimate = count / n
    return area_estimate * 4  # dividing by radius**2
```

Now let's see how fast it runs:

```
In [17]: %time calculate_pi()
```

```
    CPU times: user 361 ms, sys: 7.46 ms, total: 368 ms
Wall time: 356 ms
```

```
Out[17]: 3.136968
```

```
In [18]: %time calculate_pi()
```

```
    CPU times: user 22 ms, sys: 220 µs, total: 22.2 ms
Wall time: 13.1 ms
```

```
Out[18]: 3.141084
```

By switching parallelization on and off (selecting `True` or `False` in the `@njit` annotation), we can test the speed gain that multithreading provides on top of JIT compilation.

On our workstation, we find that parallelization increases execution speed by a factor of 2 or 3.

(If you are executing locally, you will get different numbers, depending mainly on the number of CPUs on your machine.)

Chapter 14

Pandas

14.1 Contents

- Overview 14.2
- Series 14.3
- DataFrames 14.4
- On-Line Data Sources 14.5
- Exercises 14.6
- Solutions 14.7

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

```
In [1]: !pip install --upgrade pandas-datareader
```

14.2 Overview

Pandas is a package of fast, efficient data analysis tools for Python.

Its popularity has surged in recent years, coincident with the rise of fields such as data science and machine learning.

Here's a popularity comparison over time against STATA, SAS, and dplyr courtesy of Stack Overflow Trends

Just as NumPy provides the basic array data type plus core array operations, pandas

1. defines fundamental structures for working with data and
2. endows them with methods that facilitate operations such as

- reading in data
- adjusting indices
- working with dates and time series
- sorting, grouping, re-ordering and general data munging Section **??**
- dealing with missing values, etc., etc.

More sophisticated statistical functionality is left to other packages, such as statsmodels and scikit-learn, which are built on top of pandas.

This lecture will provide a basic introduction to pandas.

Throughout the lecture, we will assume that the following imports have taken place

```
In [2]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        import requests
```

14.3 Series

Two important data types defined by pandas are `Series` and `DataFrame`.

You can think of a `Series` as a "column" of data, such as a collection of observations on a single variable.

A `DataFrame` is an object for storing related columns of data.

Let's start with Series

```
In [3]: s = pd.Series(np.random.randn(4), name='daily returns')
        s
```

```
Out[3]: 0    1.301664
        1   -2.241378
        2   -0.777007
        3    0.318483
        Name: daily returns, dtype: float64
```

Here you can imagine the indices `0, 1, 2, 3` as indexing four listed companies, and the values being daily returns on their shares.

Pandas `Series` are built on top of NumPy arrays and support many similar operations

```
In [4]: s * 100
```

```
Out[4]: 0    130.166413
        1   -224.137797
        2    -77.700727
        3     31.848293
        Name: daily returns, dtype: float64
```

```
In [5]: np.abs(s)
```

```
Out[5]: 0    1.301664
        1    2.241378
        2    0.777007
        3    0.318483
        Name: daily returns, dtype: float64
```

But `Series` provide more than NumPy arrays.

Not only do they have some additional (statistically oriented) methods

```
In [6]: s.describe()
```

```
Out[6]: count    4.000000
        mean    -0.349560
        std      1.520363
        min     -2.241378
        25%     -1.143100
        50%     -0.229262
        75%      0.564278
        max      1.301664
        Name: daily returns, dtype: float64
```

But their indices are more flexible

```
In [7]: s.index = ['AMZN', 'AAPL', 'MSFT', 'GOOG']
        s
```

```
Out[7]: AMZN    1.301664
        AAPL   -2.241378
        MSFT   -0.777007
        GOOG    0.318483
        Name: daily returns, dtype: float64
```

Viewed in this way, `Series` are like fast, efficient Python dictionaries (with the restriction that the items in the dictionary all have the same type—in this case, floats).

In fact, you can use much of the same syntax as Python dictionaries

```
In [8]: s['AMZN']
```

```
Out[8]: 1.3016641309698724
```

```
In [9]: s['AMZN'] = 0
        s
```

```
Out[9]: AMZN    0.000000
        AAPL   -2.241378
        MSFT   -0.777007
        GOOG    0.318483
        Name: daily returns, dtype: float64
```

```
In [10]: 'AAPL' in s
```

```
Out[10]: True
```

14.4 DataFrames

While a `Series` is a single column of data, a `DataFrame` is several columns, one for each variable.

In essence, a `DataFrame` in pandas is analogous to a (highly optimized) Excel spreadsheet.

Thus, it is a powerful tool for representing and analyzing data that are naturally organized into rows and columns, often with descriptive indexes for individual rows and individual columns.

Let's look at an example that reads data from the CSV file `pandas/data/test_pwt.csv` that can be downloaded here.

Here's the content of `test_pwt.csv`

```
"country","country isocode","year","POP","XRAT","tcgdp","cc","cg"
"Argentina","ARG","2000","37335.653","0.9995","295072.21869","75.716805379","5.5
"Australia","AUS","2000","19053.186","1.72483","541804.6521","67.759025993","6.7
"India","IND","2000","1006300.297","44.9416","1728144.3748","64.575551328","14.0
"Israel","ISR","2000","6114.57","4.07733","129253.89423","64.436450847","10.2666
"Malawi","MWI","2000","11801.505","59.543808333","5026.2217836","74.707624181","
"South Africa","ZAF","2000","45064.098","6.93983","227242.36949","72.718710427",
"United States","USA","2000","282171.957","1","9898700","72.347054303","6.032453
"Uruguay","URY","2000","3219.793","12.099591667","25255.961693","78.978740282","
```

Supposing you have this data saved as `test_pwt.csv` in the present working directory (type `%pwd` in Jupyter to see what this is), it can be read in as follows:

```
In [11]: df = pd.read_csv('https://raw.githubusercontent.com/QuantEcon/lecture-
source-
         py/master/source/_static/lecture_specific/pandas/data/test_pwt.csv')
         type(df)
```

```
Out[11]: pandas.core.frame.DataFrame
```

```
In [12]: df
```

```
Out[12]:          country country isocode  year          POP       XRAT        
 ↪tcgdp  \
        0      Argentina            ARG  2000    37335.653   0.999500  2.950722e+05
        1      Australia            AUS  2000    19053.186   1.724830  5.418047e+05
        2          India            IND  2000  1006300.297  44.941600  1.728144e+06
        3         Israel            ISR  2000     6114.570   4.077330  1.292539e+05
        4         Malawi            MWI  2000    11801.505  59.543808  5.026222e+03
        5   South Africa            ZAF  2000    45064.098   6.939830  2.272424e+05
        6  United States            USA  2000   282171.957   1.000000  9.898700e+06
        7        Uruguay            URY  2000     3219.793  12.099592  2.525596e+04

                  cc         cg
        0  75.716805   5.578804
        1  67.759026   6.720098
        2  64.575551  14.072206
        3  64.436451  10.266688
        4  74.707624  11.658954
        5  72.718710   5.726546
        6  72.347054   6.032454
        7  78.978740   5.108068
```

We can select particular rows using standard Python array slicing notation

```
In [13]: df[2:5]
```

```
Out[13]:   country country isocode  year          POP       XRAT         tcgdp  \
        2   India             IND  2000  1006300.297  44.941600  1.728144e+06
        3  Israel             ISR  2000     6114.570   4.077330  1.292539e+05
        4  Malawi             MWI  2000    11801.505  59.543808  5.026222e+03

                  cc         cg
        2  64.575551  14.072206
        3  64.436451  10.266688
        4  74.707624  11.658954
```

To select columns, we can pass a list containing the names of the desired columns represented as strings

```
In [14]: df[['country', 'tcgdp']]
```

```
Out[14]:          country         tcgdp
        0      Argentina  2.950722e+05
        1      Australia  5.418047e+05
        2          India  1.728144e+06
```

```
3         Israel  1.292539e+05
4         Malawi  5.026222e+03
5   South Africa  2.272424e+05
6  United States  9.898700e+06
7        Uruguay  2.525596e+04
```

To select both rows and columns using integers, the `iloc` attribute should be used with the format `.iloc[rows, columns]`

```
In [15]: df.iloc[2:5, 0:4]
```

```
Out[15]:   country country isocode  year          POP
         2   India             IND  2000  1006300.297
         3  Israel             ISR  2000     6114.570
         4  Malawi             MWI  2000    11801.505
```

To select rows and columns using a mixture of integers and labels, the `loc` attribute can be used in a similar way

```
In [16]: df.loc[df.index[2:5], ['country', 'tcgdp']]
```

```
Out[16]:   country         tcgdp
         2   India  1.728144e+06
         3  Israel  1.292539e+05
         4  Malawi  5.026222e+03
```

Let's imagine that we're only interested in population (`POP`) and total GDP (`tcgdp`).

One way to strip the data frame `df` down to only these variables is to overwrite the dataframe using the selection method described above

```
In [17]: df = df[['country', 'POP', 'tcgdp']]
         df
```

```
Out[17]:          country          POP         tcgdp
         0      Argentina    37335.653  2.950722e+05
         1      Australia    19053.186  5.418047e+05
         2          India  1006300.297  1.728144e+06
         3         Israel     6114.570  1.292539e+05
         4         Malawi    11801.505  5.026222e+03
         5   South Africa    45064.098  2.272424e+05
         6  United States   282171.957  9.898700e+06
         7        Uruguay     3219.793  2.525596e+04
```

Here the index `0, 1,..., 7` is redundant because we can use the country names as an index.

To do this, we set the index to be the `country` variable in the dataframe

```
In [18]: df = df.set_index('country')
         df
```

Out[18]:
```
                        POP         tcgdp
country
Argentina         37335.653  2.950722e+05
Australia         19053.186  5.418047e+05
India           1006300.297  1.728144e+06
Israel             6114.570  1.292539e+05
Malawi            11801.505  5.026222e+03
South Africa      45064.098  2.272424e+05
United States    282171.957  9.898700e+06
Uruguay            3219.793  2.525596e+04
```

Let's give the columns slightly better names

In [19]:
```
df.columns = 'population', 'total GDP'
df
```

Out[19]:
```
                 population     total GDP
country
Argentina         37335.653  2.950722e+05
Australia         19053.186  5.418047e+05
India           1006300.297  1.728144e+06
Israel             6114.570  1.292539e+05
Malawi            11801.505  5.026222e+03
South Africa      45064.098  2.272424e+05
United States    282171.957  9.898700e+06
Uruguay            3219.793  2.525596e+04
```

Population is in thousands, let's revert to single units

In [20]:
```
df['population'] = df['population'] * 1e3
df
```

Out[20]:
```
                   population     total GDP
country
Argentina        3.733565e+07  2.950722e+05
Australia        1.905319e+07  5.418047e+05
India            1.006300e+09  1.728144e+06
Israel           6.114570e+06  1.292539e+05
Malawi           1.180150e+07  5.026222e+03
South Africa     4.506410e+07  2.272424e+05
United States    2.821720e+08  9.898700e+06
Uruguay          3.219793e+06  2.525596e+04
```

Next, we're going to add a column showing real GDP per capita, multiplying by 1,000,000 as we go because total GDP is in millions

In [21]:
```
df['GDP percap'] = df['total GDP'] * 1e6 / df['population']
df
```

Out[21]:
```
                   population     total GDP    GDP percap
country
Argentina        3.733565e+07  2.950722e+05   7903.229085
Australia        1.905319e+07  5.418047e+05  28436.433261
```

```
India          1.006300e+09  1.728144e+06   1717.324719
Israel         6.114570e+06  1.292539e+05  21138.672749
Malawi         1.180150e+07  5.026222e+03    425.896679
South Africa   4.506410e+07  2.272424e+05   5042.647686
United States  2.821720e+08  9.898700e+06  35080.381854
Uruguay        3.219793e+06  2.525596e+04   7843.970620
```

One of the nice things about pandas `DataFrame` and `Series` objects is that they have methods for plotting and visualization that work through Matplotlib.

For example, we can easily generate a bar plot of GDP per capita

```
In [22]: ax = df['GDP percap'].plot(kind='bar')
         ax.set_xlabel('country', fontsize=12)
         ax.set_ylabel('GDP per capita', fontsize=12)
         plt.show()
```

At the moment the data frame is ordered alphabetically on the countries—let's change it to GDP per capita

```
In [23]: df = df.sort_values(by='GDP percap', ascending=False)
         df
```

```
Out[23]:                  population     total GDP    GDP percap
         country
         United States  2.821720e+08  9.898700e+06  35080.381854
```

```
Australia      1.905319e+07  5.418047e+05  28436.433261
Israel         6.114570e+06  1.292539e+05  21138.672749
Argentina      3.733565e+07  2.950722e+05   7903.229085
Uruguay        3.219793e+06  2.525596e+04   7843.970620
South Africa   4.506410e+07  2.272424e+05   5042.647686
India          1.006300e+09  1.728144e+06   1717.324719
Malawi         1.180150e+07  5.026222e+03    425.896679
```

Plotting as before now yields

```
In [24]: ax = df['GDP percap'].plot(kind='bar')
         ax.set_xlabel('country', fontsize=12)
         ax.set_ylabel('GDP per capita', fontsize=12)
         plt.show()
```

14.5 On-Line Data Sources

Python makes it straightforward to query online databases programmatically.

An important database for economists is FRED — a vast collection of time series data maintained by the St. Louis Fed.

For example, suppose that we are interested in the unemployment rate.

Via FRED, the entire series for the US civilian unemployment rate can be downloaded directly by entering this URL into your browser (note that this requires an internet connection)

`https://research.stlouisfed.org/fred2/series/UNRATE/downloaddata/UNRATE.csv`

(Equivalently, click here: https://research.stlouisfed.org/fred2/series/UNRATE/downloaddata/UNRATE.csv)

This request returns a CSV file, which will be handled by your default application for this class of files.

Alternatively, we can access the CSV file from within a Python program.

This can be done with a variety of methods.

We start with a relatively low-level method and then return to pandas.

14.5.1 Accessing Data with requests

One option is to use requests, a standard Python library for requesting data over the Internet.

To begin, try the following code on your computer

```
In [25]: r = requests.get('http://research.stlouisfed.org/fred2/series/UNRATE/
 ↪downloaddata/UNRATE
         .csv')
```

If there's no error message, then the call has succeeded.

If you do get an error, then there are two likely causes

1. You are not connected to the Internet — hopefully, this isn't the case.
2. Your machine is accessing the Internet through a proxy server, and Python isn't aware of this.

In the second case, you can either

- switch to another machine
- solve your proxy problem by reading the documentation

Assuming that all is working, you can now proceed to use the `source` object returned by the call `requests.get('http://research.stlouisfed.org/fred2/series/UNRATE/downloaddata/UN`

```
In [26]: url = 'http://research.stlouisfed.org/fred2/series/UNRATE/downloaddata/
 ↪UNRATE.csv'
         source = requests.get(url).content.decode().split("\n")
         source[0]

Out[26]: 'DATE,VALUE\r'

In [27]: source[1]

Out[27]: '1948-01-01,3.4\r'
```

```
In [28]: source[2]
```

```
Out[28]: '1948-02-01,3.8\r'
```

We could now write some additional code to parse this text and store it as an array.

But this is unnecessary — pandas' `read_csv` function can handle the task for us.

We use `parse_dates=True` so that pandas recognizes our dates column, allowing for simple date filtering

```
In [29]: data = pd.read_csv(url, index_col=0, parse_dates=True)
```

The data has been read into a pandas DataFrame called `data` that we can now manipulate in the usual way

```
In [30]: type(data)
```

```
Out[30]: pandas.core.frame.DataFrame
```

```
In [31]: data.head()  # A useful method to get a quick look at a data frame
```

```
Out[31]:             VALUE
         DATE
         1948-01-01    3.4
         1948-02-01    3.8
         1948-03-01    4.0
         1948-04-01    3.9
         1948-05-01    3.5
```

```
In [32]: pd.set_option('precision', 1)
         data.describe()  # Your output might differ slightly
```

```
Out[32]:        VALUE
         count  865.0
         mean     5.7
         std      1.6
         min      2.5
         25%      4.5
         50%      5.5
         75%      6.8
         max     10.8
```

We can also plot the unemployment rate from 2006 to 2012 as follows

```
In [33]: ax = data['2006':'2012'].plot(title='US Unemployment Rate', legend=False)
         ax.set_xlabel('year', fontsize=12)
         ax.set_ylabel('%', fontsize=12)
         plt.show()
```

Note that pandas offers many other file type alternatives.

Pandas has a wide variety of top-level methods that we can use to read, excel, json, parquet or plug straight into a database server.

14.5.2 Using pandas_datareader to Access Data

The maker of pandas has also authored a library called pandas_datareader that gives programmatic access to many data sources straight from the Jupyter notebook.

While some sources require an access key, many of the most important (e.g., FRED, OECD, EUROSTAT and the World Bank) are free to use.

For now let's work through one example of downloading and plotting data — this time from the World Bank.

The World Bank collects and organizes data on a huge range of indicators.

For example, here's some data on government debt as a ratio to GDP.

The next code example fetches the data for you and plots time series for the US and Australia

```
In [34]: from pandas_datareader import wb

         govt_debt = wb.download(indicator='GC.DOD.TOTL.GD.ZS', country=['US',
↪'AU'], start=2005,
         end=2016).stack().unstack(0)
         ind = govt_debt.index.droplevel(-1)
         govt_debt.index = ind
         ax = govt_debt.plot(lw=2)
```

```
ax.set_xlabel('year', fontsize=12)
plt.title("Government Debt to GDP (%)")
plt.show()
```

The documentation provides more details on how to access various data sources.

14.6 Exercises

14.6.1 Exercise 1

With these imports:

```
In [35]: import datetime as dt
         from pandas_datareader import data
```

Write a program to calculate the percentage price change over 2019 for the following shares:

```
In [36]: ticker_list = {'INTC': 'Intel',
                        'MSFT': 'Microsoft',
                        'IBM': 'IBM',
                        'BHP': 'BHP',
                        'TM': 'Toyota',
                        'AAPL': 'Apple',
                        'AMZN': 'Amazon',
                        'BA': 'Boeing',
                        'QCOM': 'Qualcomm',
                        'KO': 'Coca-Cola',
```

```
                    'GOOG': 'Google',
                    'SNE': 'Sony',
                    'PTR': 'PetroChina'}
```

Here's the first part of the program

```
In [37]: def read_data(ticker_list,
                       start=dt.datetime(2019, 1, 2),
                       end=dt.datetime(2019, 12, 31)):
             """
             This function reads in closing price data from Yahoo
             for each tick in the ticker_list.
             """
             ticker = pd.DataFrame()

             for tick in ticker_list:
                 prices = data.DataReader(tick, 'yahoo', start, end)
                 closing_prices = prices['Close']
                 ticker[tick] = closing_prices

             return ticker

         ticker = read_data(ticker_list)
```

Complete the program to plot the result as a bar graph like this one:

14.7 Solutions

14.7.1 Exercise 1

There are a few ways to approach this problem using Pandas to calculate the percentage change.

First, you can extract the data and perform the calculation such as:

```
In [38]: p1 = ticker.iloc[0]    #Get the first set of prices as a Series
         p2 = ticker.iloc[-1]   #Get the last set of prices as a Series
         price_change = (p2 - p1) / p1 * 100
         price_change
```

```
Out[38]: INTC    27.1
         MSFT    56.0
         IBM     16.3
         BHP     14.3
         TM      20.9
         AAPL    85.9
         AMZN    20.1
         BA       0.6
         QCOM    53.7
         KO      17.9
         GOOG    27.8
         SNE     39.6
         PTR    -17.4
         dtype: float64
```

Alternatively you can use an inbuilt method `pct_change` and configure it to perform the correct calculation using `periods` argument.

```
In [39]: change = ticker.pct_change(periods=len(ticker)-1, axis='rows')*100
         price_change = change.iloc[-1]
         price_change
```

```
Out[39]: INTC    27.1
         MSFT    56.0
         IBM     16.3
         BHP     14.3
         TM      20.9
         AAPL    85.9
         AMZN    20.1
         BA       0.6
         QCOM    53.7
         KO      17.9
         GOOG    27.8
         SNE     39.6
         PTR    -17.4
         Name: 2019-12-31 00:00:00, dtype: float64
```

Then to plot the chart

```
In [40]: price_change.sort_values(inplace=True)
         price_change = price_change.rename(index=ticker_list)
```

```
fig, ax = plt.subplots(figsize=(10,8))
ax.set_xlabel('stock', fontsize=12)
ax.set_ylabel('percentage change in price', fontsize=12)
price_change.plot(kind='bar', ax=ax)
plt.show()
```

Footnotes

[1] Wikipedia defines munging as cleaning data from one raw form into a structured, purged one.

Part III

Advanced Python Programming

Chapter 15

Writing Good Code

15.1 Contents

- Overview 15.2
- An Example of Poor Code 15.3
- Good Coding Practice 15.4
- Revisiting the Example 15.5
- Exercises 15.6
- Solutions 15.7

15.2 Overview

When computer programs are small, poorly written code is not overly costly.

But more data, more sophisticated models, and more computer power are enabling us to take on more challenging problems that involve writing longer programs.

For such programs, investment in good coding practices will pay high returns.

The main payoffs are higher productivity and faster code.

In this lecture, we review some elements of good coding practice.

We also touch on modern developments in scientific computing — such as just in time compilation — and how they affect good program design.

15.3 An Example of Poor Code

Let's have a look at some poorly written code.

The job of the code is to generate and plot time series of the simplified Solow model

$$k_{t+1} = sk_t^\alpha + (1-\delta)k_t, \quad t = 0, 1, 2, ... \tag{1}$$

Here

- k_t is capital at time t and
- s, α, δ are parameters (savings, a productivity parameter and depreciation)

For each parameterization, the code

1. sets $k_0 = 1$
2. iterates using (1) to produce a sequence $k_0, k_1, k_2 \dots, k_T$
3. plots the sequence

The plots will be grouped into three subfigures.

In each subfigure, two parameters are held fixed while another varies

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Allocate memory for time series
k = np.empty(50)

fig, axes = plt.subplots(3, 1, figsize=(6, 14))

# Trajectories with different α
δ = 0.1
s = 0.4
α = (0.25, 0.33, 0.45)

for j in range(3):
    k[0] = 1
    for t in range(49):
        k[t+1] = s * k[t]**α[j] + (1 - δ) * k[t]
    axes[0].plot(k, 'o-', label=rf"$\alpha = {α[j]},\; s = {s},\;
↪\delta={δ}$")

axes[0].grid(lw=0.2)
axes[0].set_ylim(0, 18)
axes[0].set_xlabel('time')
axes[0].set_ylabel('capital')
axes[0].legend(loc='upper left', frameon=True)

# Trajectories with different s
δ = 0.1
α = 0.33
s = (0.3, 0.4, 0.5)

for j in range(3):
    k[0] = 1
    for t in range(49):
        k[t+1] = s[j] * k[t]**α + (1 - δ) * k[t]
    axes[1].plot(k, 'o-', label=rf"$\alpha = {α},\; s = {s[j]},\;
↪\delta={δ}$")

axes[1].grid(lw=0.2)
axes[1].set_xlabel('time')
axes[1].set_ylabel('capital')
axes[1].set_ylim(0, 18)
axes[1].legend(loc='upper left', frameon=True)
```

```
# Trajectories with different δ
δ = (0.05, 0.1, 0.15)
α = 0.33
s = 0.4

for j in range(3):
    k[0] = 1
    for t in range(49):
        k[t+1] = s * k[t]**α + (1 - δ[j]) * k[t]
    axes[2].plot(k, 'o-', label=rf"$\alpha = {α},\; s = {s},\;
↪\delta={δ[j]}$")

axes[2].set_ylim(0, 18)
axes[2].set_xlabel('time')
axes[2].set_ylabel('capital')
axes[2].grid(lw=0.2)
axes[2].legend(loc='upper left', frameon=True)

plt.show()
```

$\alpha = 0.25,\ s = 0.4,\ \delta = 0.1$
$\alpha = 0.33,\ s = 0.4,\ \delta = 0.1$
$\alpha = 0.45,\ s = 0.4,\ \delta = 0.1$
capital
time

$\alpha = 0.33,\ s = 0.3,\ \delta = 0.1$
$\alpha = 0.33,\ s = 0.4,\ \delta = 0.1$
$\alpha = 0.33,\ s = 0.5,\ \delta = 0.1$
capital
time

$\alpha = 0.33,\ s = 0.4,\ \delta = 0.05$
$\alpha = 0.33,\ s = 0.4,\ \delta = 0.1$
$\alpha = 0.33,\ s = 0.4,\ \delta = 0.15$
capital
time

True, the code more or less follows PEP8.

At the same time, it's very poorly structured.

Let's talk about why that's the case, and what we can do about it.

15.4 Good Coding Practice

There are usually many different ways to write a program that accomplishes a given task.

For small programs, like the one above, the way you write code doesn't matter too much.

But if you are ambitious and want to produce useful things, you'll write medium to large programs too.

In those settings, coding style matters **a great deal**.

Fortunately, lots of smart people have thought about the best way to write code.

Here are some basic precepts.

15.4.1 Don't Use Magic Numbers

If you look at the code above, you'll see numbers like `50` and `49` and `3` scattered through the code.

These kinds of numeric literals in the body of your code are sometimes called "magic numbers".

This is not a compliment.

While numeric literals are not all evil, the numbers shown in the program above should certainly be replaced by named constants.

For example, the code above could declare the variable `time_series_length = 50`.

Then in the loops, `49` should be replaced by `time_series_length - 1`.

The advantages are:

- the meaning is much clearer throughout
- to alter the time series length, you only need to change one value

15.4.2 Don't Repeat Yourself

The other mortal sin in the code snippet above is repetition.

Blocks of logic (such as the loop to generate time series) are repeated with only minor changes.

This violates a fundamental tenet of programming: Don't repeat yourself (DRY).

- Also called DIE (duplication is evil).

Yes, we realize that you can just cut and paste and change a few symbols.

But as a programmer, your aim should be to **automate** repetition, **not** do it yourself.

More importantly, repeating the same logic in different places means that eventually one of them will likely be wrong.

If you want to know more, read the excellent summary found on this page.

We'll talk about how to avoid repetition below.

15.4.3 Minimize Global Variables

Sure, global variables (i.e., names assigned to values outside of any function or class) are convenient.

Rookie programmers typically use global variables with abandon — as we once did ourselves.

But global variables are dangerous, especially in medium to large size programs, since

- they can affect what happens in any part of your program
- they can be changed by any function

This makes it much harder to be certain about what some small part of a given piece of code actually commands.

Here's a useful discussion on the topic.

While the odd global in small scripts is no big deal, we recommend that you teach yourself to avoid them.

(We'll discuss how just below).

JIT Compilation

For scientific computing, there is another good reason to avoid global variables.

As we've seen in previous lectures, JIT compilation can generate excellent performance for scripting languages like Python.

But the task of the compiler used for JIT compilation becomes harder when global variables are present.

Put differently, the type inference required for JIT compilation is safer and more effective when variables are sandboxed inside a function.

15.4.4 Use Functions or Classes

Fortunately, we can easily avoid the evils of global variables and WET code.

- WET stands for "we enjoy typing" and is the opposite of DRY.

We can do this by making frequent use of functions or classes.

In fact, functions and classes are designed specifically to help us avoid shaming ourselves by repeating code or excessive use of global variables.

Which One, Functions or Classes?

Both can be useful, and in fact they work well with each other.

We'll learn more about these topics over time.

(Personal preference is part of the story too)

What's really important is that you use one or the other or both.

15.5 Revisiting the Example

Here's some code that reproduces the plot above with better coding style.

In [2]:
```
from itertools import product

def plot_path(ax, αs, s_vals, δs, time_series_length=50):
    """
    Add a time series plot to the axes ax for all given parameters.
    """
    k = np.empty(time_series_length)

    for (α, s, δ) in product(αs, s_vals, δs):
        k[0] = 1
        for t in range(time_series_length-1):
            k[t+1] = s * k[t]**α + (1 - δ) * k[t]
      ax.plot(k, 'o-', label=rf"$\alpha = {α},\; s = {s},\; \delta = {δ}$")

    ax.set_xlabel('time')
    ax.set_ylabel('capital')
    ax.set_ylim(0, 18)
    ax.legend(loc='upper left', frameon=True)

fig, axes = plt.subplots(3, 1, figsize=(6, 14))

# Parameters (αs, s_vals, δs)
set_one = ([0.25, 0.33, 0.45], [0.4], [0.1])
set_two = ([0.33], [0.3, 0.4, 0.5], [0.1])
set_three = ([0.33], [0.4], [0.05, 0.1, 0.15])

for (ax, params) in zip(axes, (set_one, set_two, set_three)):
    αs, s_vals, δs = params
    plot_path(ax, αs, s_vals, δs)

plt.show()
```

$\alpha = 0.25,\ s = 0.4,\ \delta = 0.1$
$\alpha = 0.33,\ s = 0.4,\ \delta = 0.1$
$\alpha = 0.45,\ s = 0.4,\ \delta = 0.1$
capital
time

$\alpha = 0.33,\ s = 0.3,\ \delta = 0.1$
$\alpha = 0.33,\ s = 0.4,\ \delta = 0.1$
$\alpha = 0.33,\ s = 0.5,\ \delta = 0.1$
capital
time

$\alpha = 0.33,\ s = 0.4,\ \delta = 0.05$
$\alpha = 0.33,\ s = 0.4,\ \delta = 0.1$
$\alpha = 0.33,\ s = 0.4,\ \delta = 0.15$
capital
time

If you inspect this code, you will see that

- it uses a function to avoid repetition.
- Global variables are quarantined by collecting them together at the end, not the start of the program.
- Magic numbers are avoided.
- The loop at the end where the actual work is done is short and relatively simple.

15.6 Exercises

15.6.1 Exercise 1

Here is some code that needs improving.

It involves a basic supply and demand problem.

Supply is given by

$$q_s(p) = \exp(\alpha p) - \beta.$$

The demand curve is

$$q_d(p) = \gamma p^{-\delta}.$$

The values α, β, γ and δ are **parameters**

The equilibrium p^* is the price such that $q_d(p) = q_s(p)$.

We can solve for this equilibrium using a root finding algorithm. Specifically, we will find the p such that $h(p) = 0$, where

$$h(p) := q_d(p) - q_s(p)$$

This yields the equilibrium price p^*. From this we get the equilibrium price by $q^* = q_s(p^*)$

The parameter values will be

- $\alpha = 0.1$
- $\beta = 1$
- $\gamma = 1$
- $\delta = 1$

In [3]:
```
from scipy.optimize import brentq

# Compute equilibrium
def h(p):
    return p**(-1) - (np.exp(0.1 * p) - 1)  # demand - supply

p_star = brentq(h, 2, 4)
q_star = np.exp(0.1 * p_star) - 1

print(f'Equilibrium price is {p_star: .2f}')
print(f'Equilibrium quantity is {q_star: .2f}')
```

```
    Equilibrium price is  2.93
Equilibrium quantity is  0.34
```

Let's also plot our results.

In [4]:
```
# Now plot
grid = np.linspace(2, 4, 100)
fig, ax = plt.subplots()

qs = np.exp(0.1 * grid) - 1
qd = grid**(-1)


ax.plot(grid, qd, 'b-', lw=2, label='demand')
ax.plot(grid, qs, 'g-', lw=2, label='supply')

ax.set_xlabel('price')
ax.set_ylabel('quantity')
ax.legend(loc='upper center')

plt.show()
```

We also want to consider supply and demand shifts.

For example, let's see what happens when demand shifts up, with γ increasing to 1.25:

In [5]:
```
# Compute equilibrium
def h(p):
    return 1.25 * p**(-1) - (np.exp(0.1 * p) - 1)

p_star = brentq(h, 2, 4)
```

```
q_star = np.exp(0.1 * p_star) - 1

print(f'Equilibrium price is {p_star: .2f}')
print(f'Equilibrium quantity is {q_star: .2f}')
```

```
   Equilibrium price is  3.25
Equilibrium quantity is  0.38
```

```
In [6]: # Now plot
        p_grid = np.linspace(2, 4, 100)
        fig, ax = plt.subplots()

        qs = np.exp(0.1 * p_grid) - 1
        qd = 1.25 * p_grid**(-1)


        ax.plot(grid, qd, 'b-', lw=2, label='demand')
        ax.plot(grid, qs, 'g-', lw=2, label='supply')

        ax.set_xlabel('price')
        ax.set_ylabel('quantity')
        ax.legend(loc='upper center')

        plt.show()
```

Now we might consider supply shifts, but you already get the idea that there's a lot of repeated code here.

Refactor and improve clarity in the code above using the principles discussed in this lecture.

15.7 Solutions

15.7.1 Exercise 1

Here's one solution, that uses a class:

In [7]:
```
class Equilibrium:

    def __init__(self, α=0.1, β=1, γ=1, δ=1):
        self.α, self.β, self.γ, self.δ = α, β, γ, δ

    def qs(self, p):
        return np.exp(self.α * p) - self.β

    def qd(self, p):
        return self.γ * p**(-self.δ)

    def compute_equilibrium(self):
        def h(p):
            return self.qd(p) - self.qs(p)
        p_star = brentq(h, 2, 4)
        q_star = np.exp(self.α * p_star) - self.β

        print(f'Equilibrium price is {p_star: .2f}')
        print(f'Equilibrium quantity is {q_star: .2f}')

    def plot_equilibrium(self):
        # Now plot
        grid = np.linspace(2, 4, 100)
        fig, ax = plt.subplots()

        ax.plot(grid, self.qd(grid), 'b-', lw=2, label='demand')
        ax.plot(grid, self.qs(grid), 'g-', lw=2, label='supply')

        ax.set_xlabel('price')
        ax.set_ylabel('quantity')
        ax.legend(loc='upper center')

        plt.show()
```

Let's create an instance at the default parameter values.

In [8]:
```
eq = Equilibrium()
```

Now we'll compute the equilibrium and plot it.

In [9]:
```
eq.compute_equilibrium()
```

```
    Equilibrium price is  2.93
Equilibrium quantity is  0.34
```

In [10]:
```
eq.plot_equilibrium()
```

One of the nice things about our refactored code is that, when we change parameters, we don't need to repeat ourselves:

```
In [11]: eq.γ = 1.25
```

```
In [12]: eq.compute_equilibrium()
```

```
    Equilibrium price is  3.25
Equilibrium quantity is  0.38
```

```
In [13]: eq.plot_equilibrium()
```

demand
supply
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
quantity
2.00
2.25
2.50
2.75
3.00
3.25
3.50
3.75
4.00
price

Chapter 16

More Language Features

16.1 Contents

- Overview 16.2
- Iterables and Iterators 16.3
- Names and Name Resolution 16.4
- Handling Errors 16.5
- Decorators and Descriptors 16.6
- Generators 16.7
- Recursive Function Calls 16.8
- Exercises 16.9
- Solutions 16.10

16.2 Overview

With this last lecture, our advice is to **skip it on first pass**, unless you have a burning desire to read it.

It's here

1. as a reference, so we can link back to it when required, and

2. for those who have worked through a number of applications, and now want to learn more about the Python language

A variety of topics are treated in the lecture, including generators, exceptions and descriptors.

16.3 Iterables and Iterators

We've already said something about iterating in Python.

Now let's look more closely at how it all works, focusing in Python's implementation of the `for` loop.

16.3.1 Iterators

Iterators are a uniform interface to stepping through elements in a collection.

Here we'll talk about using iterators—later we'll learn how to build our own.

Formally, an *iterator* is an object with a `__next__` method.

For example, file objects are iterators .

To see this, let's have another look at the US cities data, which is written to the present working directory in the following cell

```
In [1]: %%file us_cities.txt
        new york: 8244910
        los angeles: 3819702
        chicago: 2707120
        houston: 2145146
        philadelphia: 1536471
        phoenix: 1469471
        san antonio: 1359758
        san diego: 1326179
        dallas: 1223229
```

```
Overwriting us_cities.txt
```

```
In [2]: f = open('us_cities.txt')
        f.__next__()
```

```
Out[2]: 'new york: 8244910\n'
```

```
In [3]: f.__next__()
```

```
Out[3]: 'los angeles: 3819702\n'
```

We see that file objects do indeed have a `__next__` method, and that calling this method returns the next line in the file.

The next method can also be accessed via the builtin function `next()`, which directly calls this method

```
In [4]: next(f)
```

```
Out[4]: 'chicago: 2707120\n'
```

The objects returned by `enumerate()` are also iterators

```
In [5]: e = enumerate(['foo', 'bar'])
        next(e)
```

```
Out[5]: (0, 'foo')
```

```
In [6]: next(e)
```

```
Out[6]: (1, 'bar')
```

as are the reader objects from the `csv` module .

Let's create a small csv file that contains data from the NIKKEI index

```
In [7]: %%file test_table.csv
        Date,Open,High,Low,Close,Volume,Adj Close
        2009-05-21,9280.35,9286.35,9189.92,9264.15,133200,9264.15
        2009-05-20,9372.72,9399.40,9311.61,9344.64,143200,9344.64
        2009-05-19,9172.56,9326.75,9166.97,9290.29,167000,9290.29
        2009-05-18,9167.05,9167.82,8997.74,9038.69,147800,9038.69
        2009-05-15,9150.21,9272.08,9140.90,9265.02,172000,9265.02
        2009-05-14,9212.30,9223.77,9052.41,9093.73,169400,9093.73
        2009-05-13,9305.79,9379.47,9278.89,9340.49,176000,9340.49
        2009-05-12,9358.25,9389.61,9298.61,9298.61,188400,9298.61
        2009-05-11,9460.72,9503.91,9342.75,9451.98,230800,9451.98
        2009-05-08,9351.40,9464.43,9349.57,9432.83,220200,9432.83
```

```
Overwriting test_table.csv
```

```
In [8]: from csv import reader

        f = open('test_table.csv', 'r')
        nikkei_data = reader(f)
        next(nikkei_data)
```

```
Out[8]: ['Date', 'Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close']
```

```
In [9]: next(nikkei_data)
```

```
Out[9]: ['2009-05-21', '9280.35', '9286.35', '9189.92', '9264.15', '133200',
 ↪'9264.15']
```

16.3.2 Iterators in For Loops

All iterators can be placed to the right of the `in` keyword in `for` loop statements.

In fact this is how the `for` loop works: If we write

```
for x in iterator:
    <code block>
```

then the interpreter

- calls `iterator.__next__()` and binds `x` to the result
- executes the code block
- repeats until a `StopIteration` error occurs

So now you know how this magical looking syntax works

```
f = open('somefile.txt', 'r')
for line in f:
    # do something
```

The interpreter just keeps

1. calling `f.__next__()` and binding `line` to the result
2. executing the body of the loop

This continues until a `StopIteration` error occurs.

16.3.3 Iterables

You already know that we can put a Python list to the right of `in` in a `for` loop

```
In [10]: for i in ['spam', 'eggs']:
             print(i)
```

```
    spam
eggs
```

So does that mean that a list is an iterator?

The answer is no

```
In [11]: x = ['foo', 'bar']
         type(x)
```

```
Out[11]: list
```

```
In [12]: next(x)
```

```
      
↪---------------------------------------------------------------------------

      TypeError                                 Traceback (most
↪recent call last)

      <ipython-input-12-92de4e9f6b1e> in <module>
   ----> 1 next(x)


      TypeError: 'list' object is not an iterator
```

So why can we iterate over a list in a `for` loop?

The reason is that a list is *iterable* (as opposed to an iterator).

Formally, an object is iterable if it can be converted to an iterator using the built-in function `iter()`.

Lists are one such object

```
In [13]: x = ['foo', 'bar']
         type(x)
```

```
Out[13]: list
```

```
In [14]: y = iter(x)
         type(y)
```

```
Out[14]: list_iterator
```

```
In [15]: next(y)
```

```
Out[15]: 'foo'
```

```
In [16]: next(y)
```

```
Out[16]: 'bar'
```

```
In [17]: next(y)
```

```
↪---------------------------------------------------------------------------

      StopIteration                             Traceback (most
↪recent call last)

      <ipython-input-17-81b9d2f0f16a> in <module>
  ----> 1 next(y)


      StopIteration:
```

Many other objects are iterable, such as dictionaries and tuples.

Of course, not all objects are iterable

```
In [18]: iter(42)
```

```
↪---------------------------------------------------------------------------

    TypeError                                 Traceback (most
↪recent call last)

    <ipython-input-18-ef50b48e4398> in <module>
----> 1 iter(42)


    TypeError: 'int' object is not iterable
```

To conclude our discussion of `for` loops

- `for` loops work on either iterators or iterables.
- In the second case, the iterable is converted into an iterator before the loop starts.

16.3.4 Iterators and built-ins

Some built-in functions that act on sequences also work with iterables

- `max()`, `min()`, `sum()`, `all()`, `any()`

For example

```
In [19]: x = [10, -10]
         max(x)
```

```
Out[19]: 10
```

```
In [20]: y = iter(x)
         type(y)
```

```
Out[20]: list_iterator
```

```
In [21]: max(y)
```

```
Out[21]: 10
```

One thing to remember about iterators is that they are depleted by use

```
In [22]: x = [10, -10]
         y = iter(x)
         max(y)
```

```
Out[22]: 10
```

```
In [23]: max(y)
```

```
↪---------------------------------------------------------------------------

    ValueError                                Traceback (most
↪recent call last)

    <ipython-input-23-062424e6ec08> in <module>
----> 1 max(y)


    ValueError: max() arg is an empty sequence
```

16.4 Names and Name Resolution

16.4.1 Variable Names in Python

Consider the Python statement

```
In [ ]: x = 42
```

We now know that when this statement is executed, Python creates an object of type `int` in your computer's memory, containing

- the value `42`
- some associated attributes

But what is `x` itself?

In Python, `x` is called a *name*, and the statement `x = 42` *binds* the name `x` to the integer object we have just discussed.

Under the hood, this process of binding names to objects is implemented as a dictionary—more about this in a moment.

There is no problem binding two or more names to the one object, regardless of what that object is

```
In [24]: def f(string):      # Create a function called f
             print(string)   # that prints any string it's passed

         g = f
         id(g) == id(f)
```

```
Out[24]: True
```

```
In [25]: g('test')
```

```
         test
```

In the first step, a function object is created, and the name `f` is bound to it.

After binding the name `g` to the same object, we can use it anywhere we would use `f`.

What happens when the number of names bound to an object goes to zero?

Here's an example of this situation, where the name `x` is first bound to one object and then rebound to another

```
In [26]: x = 'foo'
         id(x)
```

```
Out[26]: 140412414118464
```

```
In [27]: x = 'bar'  # No names bound to the first object
```

What happens here is that the first object is garbage collected.

In other words, the memory slot that stores that object is deallocated, and returned to the operating system.

16.4.2 Namespaces

Recall from the preceding discussion that the statement

```
In [28]: x = 42
```

binds the name `x` to the integer object on the right-hand side.

We also mentioned that this process of binding `x` to the correct object is implemented as a dictionary.

This dictionary is called a *namespace*.

Definition: A namespace is a symbol table that maps names to objects in memory.

Python uses multiple namespaces, creating them on the fly as necessary .

For example, every time we import a module, Python creates a namespace for that module.

To see this in action, suppose we write a script `math2.py` with a single line

```
In [29]: %%file math2.py
         pi = 'foobar'
```

```
Overwriting math2.py
```

Now we start the Python interpreter and import it

```
In [30]: import math2
```

Next let's import the `math` module from the standard library

```
In [31]: import math
```

Both of these modules have an attribute called `pi`

```
In [32]: math.pi

Out[32]: 3.141592653589793

In [33]: math2.pi

Out[33]: 'foobar'
```

These two different bindings of `pi` exist in different namespaces, each one implemented as a dictionary.

We can look at the dictionary directly, using `module_name.__dict__`

```
In [34]: import math

         math.__dict__.items()

Out[34]: dict_items([('__name__', 'math'), ('__doc__', 'This module is always
 ↪available.  It
         provides access to the\nmathematical functions defined by the C standard.
 ↪'),
         ('__package__', ''), ('__loader__', <_frozen_importlib_external.
 ↪ExtensionFileLoader
         object at 0x7fb450919470>), ('__spec__', ModuleSpec(name='math',
         loader=<_frozen_importlib_external.ExtensionFileLoader object at
 ↪0x7fb450919470>,
        origin='/home/ubuntu/anaconda3/lib/python3.7/lib-dynload/math.cpython-37m-
x86_64-linux-
         gnu.so')), ('acos', <built-in function acos>), ('acosh', <built-in
 ↪function acosh>),
         ('asin', <built-in function asin>), ('asinh', <built-in function asinh>),
 ↪('atan',
         <built-in function atan>), ('atan2', <built-in function atan2>),
 ↪('atanh', <built-in
         function atanh>), ('ceil', <built-in function ceil>), ('copysign', <built-
in function
         copysign>), ('cos', <built-in function cos>), ('cosh', <built-in function
 ↪cosh>),
         ('degrees', <built-in function degrees>), ('erf', <built-in function
 ↪erf>), ('erfc',
         <built-in function erfc>), ('exp', <built-in function exp>), ('expm1',
 ↪<built-in
         function expm1>), ('fabs', <built-in function fabs>), ('factorial',
 ↪<built-in function
         factorial>), ('floor', <built-in function floor>), ('fmod', <built-in
 ↪function fmod>),
         ('frexp', <built-in function frexp>), ('fsum', <built-in function fsum>),
 ↪('gamma',
         <built-in function gamma>), ('gcd', <built-in function gcd>), ('hypot',
 ↪<built-in
```

```
        function hypot>), ('isclose', <built-in function isclose>), ('isfinite',
↪<built-in
        function isfinite>), ('isinf', <built-in function isinf>), ('isnan',
↪<built-in function
        isnan>), ('ldexp', <built-in function ldexp>), ('lgamma', <built-in
↪function lgamma>),
        ('log', <built-in function log>), ('log1p', <built-in function log1p>),
↪('log10',
        <built-in function log10>), ('log2', <built-in function log2>), ('modf',
↪<built-in
        function modf>), ('pow', <built-in function pow>), ('radians', <built-in
↪function
        radians>), ('remainder', <built-in function remainder>), ('sin', <built-
in function
        sin>), ('sinh', <built-in function sinh>), ('sqrt', <built-in function
↪sqrt>), ('tan',
        <built-in function tan>), ('tanh', <built-in function tanh>), ('trunc',
↪<built-in
        function trunc>), ('pi', 3.141592653589793), ('e', 2.718281828459045),
↪('tau',
        6.283185307179586), ('inf', inf), ('nan', nan), ('__file__',
        '/home/ubuntu/anaconda3/lib/python3.7/lib-dynload/math.cpython-37m-x86_64-
linux-
        gnu.so')])
```

```
In [35]: import math2

         math2.__dict__.items()
```

```
Out[35]: dict_items([('__name__', 'math2'), ('__doc__', None), ('__package__', ''),
         ('__loader__', <_frozen_importlib_external.SourceFileLoader object at
↪0x7fb4509e7ac8>),
         ('__spec__', ModuleSpec(name='math2',
         loader=<_frozen_importlib_external.SourceFileLoader object at
↪0x7fb4509e7ac8>,
         origin='/home/ubuntu/repos/lecture-source-py/_build/jupyterpdf/executed/
↪math2.py')),
         ('__file__', '/home/ubuntu/repos/lecture-source-
         py/_build/jupyterpdf/executed/math2.py'), ('__cached__', '/home/ubuntu/
↪repos/lecture-
         source-py/_build/jupyterpdf/executed/__pycache__/math2.cpython-37.pyc'),
         ('__builtins__', {'__name__': 'builtins', '__doc__': "Built-in functions,
↪exceptions,
         and other objects.\n\nNoteworthy: None is the `nil' object; Ellipsis
↪represents `…' in
         slices.", '__package__': '', '__loader__': <class '_frozen_importlib.
↪BuiltinImporter'>,
         '__spec__': ModuleSpec(name='builtins', loader=<class
         '_frozen_importlib.BuiltinImporter'>), '__build_class__': <built-in
↪function
         __build_class__>, '__import__': <built-in function __import__>, 'abs':
↪<built-in
         function abs>, 'all': <built-in function all>, 'any': <built-in function
↪any>, 'ascii':
         <built-in function ascii>, 'bin': <built-in function bin>, 'breakpoint':
↪<built-in
```

```
        function breakpoint>, 'callable': <built-in function callable>, 'chr':
↪<built-in
        function chr>, 'compile': <built-in function compile>, 'delattr': <built-
in function
        delattr>, 'dir': <built-in function dir>, 'divmod': <built-in function
↪divmod>, 'eval':
        <built-in function eval>, 'exec': <built-in function exec>, 'format':
↪<built-in function
        format>, 'getattr': <built-in function getattr>, 'globals': <built-in
↪function globals>,
        'hasattr': <built-in function hasattr>, 'hash': <built-in function hash>,
↪'hex': <built-
        in function hex>, 'id': <built-in function id>, 'input': <bound method
↪Kernel.raw_input
        of <ipykernel.ipkernel.IPythonKernel object at 0x7fb44e5ddb00>>,
↪'isinstance': <built-in
        function isinstance>, 'issubclass': <built-in function issubclass>,
↪'iter': <built-in
        function iter>, 'len': <built-in function len>, 'locals': <built-in
↪function locals>,
        'max': <built-in function max>, 'min': <built-in function min>, 'next':
↪<built-in
        function next>, 'oct': <built-in function oct>, 'ord': <built-in function
↪ord>, 'pow':
        <built-in function pow>, 'print': <built-in function print>, 'repr':
↪<built-in function
        repr>, 'round': <built-in function round>, 'setattr': <built-in function
↪setattr>,
        'sorted': <built-in function sorted>, 'sum': <built-in function sum>,
↪'vars': <built-in
        function vars>, 'None': None, 'Ellipsis': Ellipsis, 'NotImplemented':
↪NotImplemented,
        'False': False, 'True': True, 'bool': <class 'bool'>, 'memoryview': <class
        'memoryview'>, 'bytearray': <class 'bytearray'>, 'bytes': <class 'bytes'>,
        'classmethod': <class 'classmethod'>, 'complex': <class 'complex'>,
↪'dict': <class
        'dict'>, 'enumerate': <class 'enumerate'>, 'filter': <class 'filter'>,
↪'float': <class
        'float'>, 'frozenset': <class 'frozenset'>, 'property': <class
↪'property'>, 'int':
        <class 'int'>, 'list': <class 'list'>, 'map': <class 'map'>, 'object':
↪<class 'object'>,
        'range': <class 'range'>, 'reversed': <class 'reversed'>, 'set': <class
↪'set'>, 'slice':
        <class 'slice'>, 'staticmethod': <class 'staticmethod'>, 'str': <class
↪'str'>, 'super':
        <class 'super'>, 'tuple': <class 'tuple'>, 'type': <class 'type'>, 'zip':
↪<class 'zip'>,
        '__debug__': True, 'BaseException': <class 'BaseException'>, 'Exception':
↪<class
        'Exception'>, 'TypeError': <class 'TypeError'>, 'StopAsyncIteration':
↪<class
        'StopAsyncIteration'>, 'StopIteration': <class 'StopIteration'>,
↪'GeneratorExit': <class
        'GeneratorExit'>, 'SystemExit': <class 'SystemExit'>, 'KeyboardInterrupt':
↪ <class
```

```
        'KeyboardInterrupt'>, 'ImportError': <class 'ImportError'>,
↪'ModuleNotFoundError':
        <class 'ModuleNotFoundError'>, 'OSError': <class 'OSError'>,
↪'EnvironmentError': <class
        'OSError'>, 'IOError': <class 'OSError'>, 'EOFError': <class 'EOFError'>,
        'RuntimeError': <class 'RuntimeError'>, 'RecursionError': <class
↪'RecursionError'>,
        'NotImplementedError': <class 'NotImplementedError'>, 'NameError': <class
↪'NameError'>,
        'UnboundLocalError': <class 'UnboundLocalError'>, 'AttributeError': <class
        'AttributeError'>, 'SyntaxError': <class 'SyntaxError'>,
↪'IndentationError': <class
        'IndentationError'>, 'TabError': <class 'TabError'>, 'LookupError': <class
        'LookupError'>, 'IndexError': <class 'IndexError'>, 'KeyError': <class
↪'KeyError'>,
        'ValueError': <class 'ValueError'>, 'UnicodeError': <class 'UnicodeError'>,
        'UnicodeEncodeError': <class 'UnicodeEncodeError'>, 'UnicodeDecodeError':
↪<class
        'UnicodeDecodeError'>, 'UnicodeTranslateError': <class
↪'UnicodeTranslateError'>,
        'AssertionError': <class 'AssertionError'>, 'ArithmeticError': <class
        'ArithmeticError'>, 'FloatingPointError': <class 'FloatingPointError'>,
↪'OverflowError':
        <class 'OverflowError'>, 'ZeroDivisionError': <class 'ZeroDivisionError'>,
        'SystemError': <class 'SystemError'>, 'ReferenceError': <class
↪'ReferenceError'>,
        'MemoryError': <class 'MemoryError'>, 'BufferError': <class
↪'BufferError'>, 'Warning':
        <class 'Warning'>, 'UserWarning': <class 'UserWarning'>,
↪'DeprecationWarning': <class
        'DeprecationWarning'>, 'PendingDeprecationWarning': <class
↪'PendingDeprecationWarning'>,
        'SyntaxWarning': <class 'SyntaxWarning'>, 'RuntimeWarning': <class
↪'RuntimeWarning'>,
        'FutureWarning': <class 'FutureWarning'>, 'ImportWarning': <class
↪'ImportWarning'>,
        'UnicodeWarning': <class 'UnicodeWarning'>, 'BytesWarning': <class
↪'BytesWarning'>,
        'ResourceWarning': <class 'ResourceWarning'>, 'ConnectionError': <class
        'ConnectionError'>, 'BlockingIOError': <class 'BlockingIOError'>,
↪'BrokenPipeError':
        <class 'BrokenPipeError'>, 'ChildProcessError': <class
↪'ChildProcessError'>,
        'ConnectionAbortedError': <class 'ConnectionAbortedError'>,
↪'ConnectionRefusedError':
        <class 'ConnectionRefusedError'>, 'ConnectionResetError': <class
        'ConnectionResetError'>, 'FileExistsError': <class 'FileExistsError'>,
        'FileNotFoundError': <class 'FileNotFoundError'>, 'IsADirectoryError':
↪<class
        'IsADirectoryError'>, 'NotADirectoryError': <class 'NotADirectoryError'>,
        'InterruptedError': <class 'InterruptedError'>, 'PermissionError': <class
        'PermissionError'>, 'ProcessLookupError': <class 'ProcessLookupError'>,
↪'TimeoutError':
        <class 'TimeoutError'>, 'open': <built-in function open>, 'copyright':
↪Copyright (c)
        2001-2019 Python Software Foundation.
```

```
        All Rights Reserved.

        Copyright (c) 2000 BeOpen.com.
        All Rights Reserved.

        Copyright (c) 1995-2001 Corporation for National Research Initiatives.
        All Rights Reserved.

        Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
        All Rights Reserved., 'credits':     Thanks to CWI, CNRI, BeOpen.com,
↪Zope Corporation
        and a cast of thousands
            for supporting Python development.  See www.python.org for more
↪information.,
        'license': Type license() to see the full license text, 'help': Type
↪help() for
        interactive help, or help(object) for help about object., '__IPYTHON__':
↪True,
        'display': <function display at 0x7fb450860d08>, 'get_ipython': <bound
↪method
        InteractiveShell.get_ipython of <ipykernel.zmqshell.ZMQInteractiveShell
↪object at
        0x7fb44e5dd828>>}), ('pi', 'foobar')])
```

As you know, we access elements of the namespace using the dotted attribute notation

```
In [36]: math.pi
```

```
Out[36]: 3.141592653589793
```

In fact this is entirely equivalent to `math.__dict__['pi']`

```
In [37]: math.__dict__['pi'] == math.pi
```

```
Out[37]: True
```

16.4.3 Viewing Namespaces

As we saw above, the `math` namespace can be printed by typing `math.__dict__`.

Another way to see its contents is to type `vars(math)`

```
In [38]: vars(math).items()
```

```
Out[38]: dict_items([('__name__', 'math'), ('__doc__', 'This module is always
↪available.  It
        provides access to the\nmathematical functions defined by the C standard.
↪'),
        ('__package__', ''), ('__loader__', <_frozen_importlib_external.
↪ExtensionFileLoader
        object at 0x7fb450919470>), ('__spec__', ModuleSpec(name='math',
        loader=<_frozen_importlib_external.ExtensionFileLoader object at
↪0x7fb450919470>,
```

```
        origin='/home/ubuntu/anaconda3/lib/python3.7/lib-dynload/math.cpython-37m-
x86_64-linux-
        gnu.so')), ('acos', <built-in function acos>), ('acosh', <built-in
↪function acosh>),
        ('asin', <built-in function asin>), ('asinh', <built-in function asinh>),
↪('atan',
        <built-in function atan>), ('atan2', <built-in function atan2>),
↪('atanh', <built-in
        function atanh>), ('ceil', <built-in function ceil>), ('copysign', <built-
in function
        copysign>), ('cos', <built-in function cos>), ('cosh', <built-in function
↪cosh>),
        ('degrees', <built-in function degrees>), ('erf', <built-in function
↪erf>), ('erfc',
        <built-in function erfc>), ('exp', <built-in function exp>), ('expm1',
↪<built-in
        function expm1>), ('fabs', <built-in function fabs>), ('factorial',
↪<built-in function
        factorial>), ('floor', <built-in function floor>), ('fmod', <built-in
↪function fmod>),
        ('frexp', <built-in function frexp>), ('fsum', <built-in function fsum>),
↪('gamma',
        <built-in function gamma>), ('gcd', <built-in function gcd>), ('hypot',
↪<built-in
        function hypot>), ('isclose', <built-in function isclose>), ('isfinite',
↪<built-in
        function isfinite>), ('isinf', <built-in function isinf>), ('isnan',
↪<built-in function
        isnan>), ('ldexp', <built-in function ldexp>), ('lgamma', <built-in
↪function lgamma>),
        ('log', <built-in function log>), ('log1p', <built-in function log1p>),
↪('log10',
        <built-in function log10>), ('log2', <built-in function log2>), ('modf',
↪<built-in
        function modf>), ('pow', <built-in function pow>), ('radians', <built-in
↪function
        radians>), ('remainder', <built-in function remainder>), ('sin', <built-
in function
        sin>), ('sinh', <built-in function sinh>), ('sqrt', <built-in function
↪sqrt>), ('tan',
        <built-in function tan>), ('tanh', <built-in function tanh>), ('trunc',
↪<built-in
        function trunc>), ('pi', 3.141592653589793), ('e', 2.718281828459045),
↪('tau',
        6.283185307179586), ('inf', inf), ('nan', nan), ('__file__',
        '/home/ubuntu/anaconda3/lib/python3.7/lib-dynload/math.cpython-37m-x86_64-
linux-
        gnu.so')])
```

If you just want to see the names, you can type

```
In [39]: dir(math)[0:10]
```

```
Out[39]: ['__doc__',
          '__file__',
```

```
'__loader__',
'__name__',
'__package__',
'__spec__',
'acos',
'acosh',
'asin',
'asinh']
```

Notice the special names `__doc__` and `__name__`.

These are initialized in the namespace when any module is imported

- `__doc__` is the doc string of the module
- `__name__` is the name of the module

```
In [40]: print(math.__doc__)
```

```
    This module is always available.  It provides access to the
mathematical functions defined by the C standard.
```

```
In [41]: math.__name__
```

```
Out[41]: 'math'
```

16.4.4 Interactive Sessions

In Python, **all** code executed by the interpreter runs in some module.

What about commands typed at the prompt?

These are also regarded as being executed within a module — in this case, a module called `__main__`.

To check this, we can look at the current module name via the value of `__name__` given at the prompt

```
In [42]: print(__name__)
```

```
__main__
```

When we run a script using IPython's `run` command, the contents of the file are executed as part of `__main__` too.

To see this, let's create a file `mod.py` that prints its own `__name__` attribute

```
In [43]: %%file mod.py
         print(__name__)
```

```
Overwriting mod.py
```

Now let's look at two different ways of running it in IPython

```
In [44]: import mod  # Standard import

         mod
```

```
In [45]: %run mod.py  # Run interactively

         __main__
```

In the second case, the code is executed as part of `__main__`, so `__name__` is equal to `__main__`.

To see the contents of the namespace of `__main__` we use `vars()` rather than `vars(__main__)` .

If you do this in IPython, you will see a whole lot of variables that IPython needs, and has initialized when you started up your session.

If you prefer to see only the variables you have initialized, use `whos`

```
In [46]: x = 2
         y = 3

         import numpy as np

         %whos
```

```
    Variable      Type                          Data/Info
-----------------------------------------------------
e             enumerate                     <enumerate object at 0x7fb44c4730d8>
f             function                      <function f at 0x7fb44c3dfc80>
g             function                      <function f at 0x7fb44c3dfc80>
i             str                           eggs
math          module                        <module 'math' from
'/hom<…>37m-x86_64-linux-gnu.so'>
math2         module                        <module 'math2' from
'/ho<…>erpdf/executed/math2.py'>
mod           module                        <module 'mod' from
'/home<…>yterpdf/executed/mod.py'>
nikkei_data   reader                        <_csv.reader object at 0x7fb44c4ebf98>
np            module                        <module 'numpy' from
'/ho<…>kages/numpy/__init__.py'>
reader        builtin_function_or_method    <built-in function reader>
x             int                           2
y             int                           3
```

16.4.5 The Global Namespace

Python documentation often makes reference to the "global namespace".

The global namespace is *the namespace of the module currently being executed.*

For example, suppose that we start the interpreter and begin making assignments .

We are now working in the module `__main__`, and hence the namespace for `__main__` is the global namespace.

Next, we import a module called `amodule`

```
import amodule
```

At this point, the interpreter creates a namespace for the module `amodule` and starts executing commands in the module.

While this occurs, the namespace `amodule.__dict__` is the global namespace.

Once execution of the module finishes, the interpreter returns to the module from where the import statement was made.

In this case it's `__main__`, so the namespace of `__main__` again becomes the global namespace.

16.4.6 Local Namespaces

Important fact: When we call a function, the interpreter creates a *local namespace* for that function, and registers the variables in that namespace.

The reason for this will be explained in just a moment.

Variables in the local namespace are called *local variables*.

After the function returns, the namespace is deallocated and lost.

While the function is executing, we can view the contents of the local namespace with `locals()`.

For example, consider

```
In [47]: def f(x):
             a = 2
             print(locals())
             return a * x
```

Now let's call the function

```
In [48]: f(1)

         {'x': 1, 'a': 2}

Out[48]: 2
```

You can see the local namespace of `f` before it is destroyed.

16.4.7 The `__builtins__` Namespace

We have been using various built-in functions, such as `max()`, `dir()`, `str()`, `list()`, `len()`, `range()`, `type()`, etc.

How does access to these names work?

- These definitions are stored in a module called `__builtin__`.
- They have there own namespace called `__builtins__`.

```
In [49]: dir()[0:10]

Out[49]: ['In', 'Out', '_', '_11', '_13', '_14', '_15', '_16', '_19', '_2']

In [50]: dir(__builtins__)[0:10]

Out[50]: ['ArithmeticError',
          'AssertionError',
          'AttributeError',
          'BaseException',
          'BlockingIOError',
          'BrokenPipeError',
          'BufferError',
          'BytesWarning',
          'ChildProcessError',
          'ConnectionAbortedError']
```

We can access elements of the namespace as follows

```
In [51]: __builtins__.max

Out[51]: <function max>
```

But `__builtins__` is special, because we can always access them directly as well

```
In [52]: max

Out[52]: <function max>

In [53]: __builtins__.max == max

Out[53]: True
```

The next section explains how this works …

16.4.8 Name Resolution

Namespaces are great because they help us organize variable names.

(Type `import this` at the prompt and look at the last item that's printed)

However, we do need to understand how the Python interpreter works with multiple namespaces .

At any point of execution, there are in fact at least two namespaces that can be accessed directly.

("Accessed directly" means without using a dot, as in `pi` rather than `math.pi`)

These namespaces are

- The global namespace (of the module being executed)
- The builtin namespace

If the interpreter is executing a function, then the directly accessible namespaces are

- The local namespace of the function
- The global namespace (of the module being executed)
- The builtin namespace

Sometimes functions are defined within other functions, like so

```
In [54]: def f():
             a = 2
             def g():
                 b = 4
                 print(a * b)
             g()
```

Here `f` is the *enclosing function* for `g`, and each function gets its own namespaces.

Now we can give the rule for how namespace resolution works:

The order in which the interpreter searches for names is

1. the local namespace (if it exists)
2. the hierarchy of enclosing namespaces (if they exist)
3. the global namespace
4. the builtin namespace

If the name is not in any of these namespaces, the interpreter raises a `NameError`.

This is called the **LEGB rule** (local, enclosing, global, builtin).

Here's an example that helps to illustrate .

Consider a script `test.py` that looks as follows

```
In [55]: %%file test.py
         def g(x):
             a = 1
             x = x + a
             return x

         a = 0
         y = g(10)
         print("a = ", a, "y = ", y)
```

```
Overwriting test.py
```

What happens when we run this script?

```
In [56]: %run test.py
```

```
a =  0 y =  11
```

```
In [57]: x
```

```
Out[57]: 2
```

First,

- The global namespace `{}` is created.
- The function object is created, and `g` is bound to it within the global namespace.
- The name `a` is bound to `0`, again in the global namespace.

Next `g` is called via `y = g(10)`, leading to the following sequence of actions

- The local namespace for the function is created.
- Local names `x` and `a` are bound, so that the local namespace becomes `{'x': 10, 'a': 1}`.
- Statement `x = x + a` uses the local `a` and local `x` to compute `x + a`, and binds local name `x` to the result.
- This value is returned, and `y` is bound to it in the global namespace.
- Local `x` and `a` are discarded (and the local namespace is deallocated).

Note that the global `a` was not affected by the local `a`.

16.4.9 Mutable Versus Immutable Parameters

This is a good time to say a little more about mutable vs immutable objects.

Consider the code segment

```
In [58]: def f(x):
             x = x + 1
             return x

         x = 1
         print(f(x), x)
```

```
2 1
```

We now understand what will happen here: The code prints `2` as the value of `f(x)` and `1` as the value of `x`.

First `f` and `x` are registered in the global namespace.

The call `f(x)` creates a local namespace and adds `x` to it, bound to `1`.

Next, this local `x` is rebound to the new integer object `2`, and this value is returned.

None of this affects the global `x`.

However, it's a different story when we use a **mutable** data type such as a list

```
In [59]: def f(x):
             x[0] = x[0] + 1
```

```
    return x

x = [1]
print(f(x), x)
```

```
[2] [2]
```

This prints `[2]` as the value of `f(x)` and *same* for `x`.

Here's what happens

- `f` is registered as a function in the global namespace
- `x` bound to `[1]` in the global namespace
- The call `f(x)`
  - Creates a local namespace
  - Adds `x` to local namespace, bound to `[1]`
  - The list `[1]` is modified to `[2]`
  - Returns the list `[2]`
  - The local namespace is deallocated, and local `x` is lost
- Global `x` has been modified

16.5 Handling Errors

Sometimes it's possible to anticipate errors as we're writing code.

For example, the unbiased sample variance of sample $y_1, \dots, y_n$ is defined as

$$s^2 := \frac{1}{n-1} \sum_{i=1}^{n} (y_i - \bar{y})^2 \qquad \bar{y} = \text{ sample mean}$$

This can be calculated in NumPy using `np.var`.

But if you were writing a function to handle such a calculation, you might anticipate a divide-by-zero error when the sample size is one.

One possible action is to do nothing — the program will just crash, and spit out an error message.

But sometimes it's worth writing your code in a way that anticipates and deals with runtime errors that you think might arise.

Why?

- Because the debugging information provided by the interpreter is often less useful than the information on possible errors you have in your head when writing code.
- Because errors causing execution to stop are frustrating if you're in the middle of a large computation.
- Because it's reduces confidence in your code on the part of your users (if you are writing for others).

16.5.1 Assertions

A relatively easy way to handle checks is with the `assert` keyword.

For example, pretend for a moment that the `np.var` function doesn't exist and we need to write our own

```
In [60]: def var(y):
             n = len(y)
             assert n > 1, 'Sample size must be greater than one.'
             return np.sum((y - y.mean())**2) / float(n-1)
```

If we run this with an array of length one, the program will terminate and print our error message

```
In [61]: var([1])
```

```
↪---------------------------------------------------------------------------

      AssertionError                            Traceback (most
↪recent call last)

      <ipython-input-61-8419b6ab38ec> in <module>
  ----> 1 var([1])


      <ipython-input-60-e6ffb16a7098> in var(y)
        1 def var(y):
        2     n = len(y)
  ----> 3     assert n > 1, 'Sample size must be greater than one.'
        4     return np.sum((y - y.mean())**2) / float(n-1)


      AssertionError: Sample size must be greater than one.
```

The advantage is that we can

- fail early, as soon as we know there will be a problem
- supply specific information on why a program is failing

16.5.2 Handling Errors During Runtime

The approach used above is a bit limited, because it always leads to termination.

Sometimes we can handle errors more gracefully, by treating special cases.

Let's look at how this is done.

Exceptions

Here's an example of a common error type

```
In [62]: def f:
```

```
  File "<ipython-input-62-262a7e387ba5>", line 1
def f:
     ^
SyntaxError: invalid syntax
```

Since illegal syntax cannot be executed, a syntax error terminates execution of the program.

Here's a different kind of error, unrelated to syntax

```
In [63]: 1 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-63-bc757c3fda29> in <module>
----> 1 1 / 0


ZeroDivisionError: division by zero
```

Here's another

```
In [64]: x1 = y1
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

<ipython-input-64-a7b8d65e9e45> in <module>
----> 1 x1 = y1


NameError: name 'y1' is not defined
```

And another

```
In [65]: 'foo' + 6
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-65-216809d6e6fe> in <module>
----> 1 'foo' + 6

TypeError: can only concatenate str (not "int") to str
```

And another

```
In [66]: X = []
         x = X[0]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-66-082a18d7a0aa> in <module>
      1 X = []
----> 2 x = X[0]

IndexError: list index out of range
```

On each occasion, the interpreter informs us of the error type

- `NameError`, `TypeError`, `IndexError`, `ZeroDivisionError`, etc.

In Python, these errors are called *exceptions*.

Catching Exceptions

We can catch and deal with exceptions using `try` – `except` blocks.

Here's a simple example

```
In [67]: def f(x):
             try:
                 return 1.0 / x
```

```
        except ZeroDivisionError:
            print('Error: division by zero.  Returned None')
        return None
```

When we call `f` we get the following output

```
In [68]: f(2)
```

```
Out[68]: 0.5
```

```
In [69]: f(0)
```

```
Error: division by zero.  Returned None
```

```
In [70]: f(0.0)
```

```
Error: division by zero.  Returned None
```

The error is caught and execution of the program is not terminated.

Note that other error types are not caught.

If we are worried the user might pass in a string, we can catch that error too

```
In [71]: def f(x):
             try:
                 return 1.0 / x
             except ZeroDivisionError:
                 print('Error: Division by zero.  Returned None')
             except TypeError:
                 print('Error: Unsupported operation.  Returned None')
             return None
```

Here's what happens

```
In [72]: f(2)
```

```
Out[72]: 0.5
```

```
In [73]: f(0)
```

```
Error: Division by zero.  Returned None
```

```
In [74]: f('foo')
```

```
Error: Unsupported operation.  Returned None
```

If we feel lazy we can catch these errors together

```
In [75]: def f(x):
             try:
                 return 1.0 / x
             except (TypeError, ZeroDivisionError):
                 print('Error: Unsupported operation.  Returned None')
             return None
```

Here's what happens

```
In [76]: f(2)

Out[76]: 0.5

In [77]: f(0)

           Error: Unsupported operation.  Returned None


In [78]: f('foo')

           Error: Unsupported operation.  Returned None
```

If we feel extra lazy we can catch all error types as follows

```
In [79]: def f(x):
             try:
                 return 1.0 / x
             except:
                 print('Error.  Returned None')
             return None
```

In general it's better to be specific.

16.6 Decorators and Descriptors

Let's look at some special syntax elements that are routinely used by Python developers.

You might not need the following concepts immediately, but you will see them in other people's code.

Hence you need to understand them at some stage of your Python education.

16.6.1 Decorators

Decorators are a bit of syntactic sugar that, while easily avoided, have turned out to be popular.

It's very easy to say what decorators do.

On the other hand it takes a bit of effort to explain *why* you might use them.

An Example

Suppose we are working on a program that looks something like this

In [80]:
```
import numpy as np

def f(x):
    return np.log(np.log(x))

def g(x):
    return np.sqrt(42 * x)

# Program continues with various calculations using f and g
```

Now suppose there's a problem: occasionally negative numbers get fed to `f` and `g` in the calculations that follow.

If you try it, you'll see that when these functions are called with negative numbers they return a NumPy object called `nan` .

This stands for "not a number" (and indicates that you are trying to evaluate a mathematical function at a point where it is not defined).

Perhaps this isn't what we want, because it causes other problems that are hard to pick up later on.

Suppose that instead we want the program to terminate whenever this happens, with a sensible error message.

This change is easy enough to implement

In [81]:
```
import numpy as np

def f(x):
    assert x >= 0, "Argument must be nonnegative"
    return np.log(np.log(x))

def g(x):
    assert x >= 0, "Argument must be nonnegative"
    return np.sqrt(42 * x)

# Program continues with various calculations using f and g
```

Notice however that there is some repetition here, in the form of two identical lines of code.

Repetition makes our code longer and harder to maintain, and hence is something we try hard to avoid.

Here it's not a big deal, but imagine now that instead of just `f` and `g`, we have 20 such functions that we need to modify in exactly the same way.

This means we need to repeat the test logic (i.e., the `assert` line testing nonnegativity) 20 times.

The situation is still worse if the test logic is longer and more complicated.

In this kind of scenario the following approach would be neater

```
In [82]: import numpy as np

         def check_nonneg(func):
             def safe_function(x):
                 assert x >= 0, "Argument must be nonnegative"
                 return func(x)
             return safe_function

         def f(x):
             return np.log(np.log(x))

         def g(x):
             return np.sqrt(42 * x)

         f = check_nonneg(f)
         g = check_nonneg(g)
         # Program continues with various calculations using f and g
```

This looks complicated so let's work through it slowly.

To unravel the logic, consider what happens when we say `f = check_nonneg(f)`.

This calls the function `check_nonneg` with parameter `func` set equal to `f`.

Now `check_nonneg` creates a new function called `safe_function` that verifies `x` as non-negative and then calls `func` on it (which is the same as `f`).

Finally, the global name `f` is then set equal to `safe_function`.

Now the behavior of `f` is as we desire, and the same is true of `g`.

At the same time, the test logic is written only once.

Enter Decorators

The last version of our code is still not ideal.

For example, if someone is reading our code and wants to know how `f` works, they will be looking for the function definition, which is

```
In [83]: def f(x):
             return np.log(np.log(x))
```

They may well miss the line `f = check_nonneg(f)`.

For this and other reasons, decorators were introduced to Python.

With decorators, we can replace the lines

```
In [84]: def f(x):
             return np.log(np.log(x))

         def g(x):
             return np.sqrt(42 * x)

         f = check_nonneg(f)
         g = check_nonneg(g)
```

with

```
In [85]: @check_nonneg
         def f(x):
             return np.log(np.log(x))

         @check_nonneg
         def g(x):
             return np.sqrt(42 * x)
```

These two pieces of code do exactly the same thing.

If they do the same thing, do we really need decorator syntax?

Well, notice that the decorators sit right on top of the function definitions.

Hence anyone looking at the definition of the function will see them and be aware that the function is modified.

In the opinion of many people, this makes the decorator syntax a significant improvement to the language.

16.6.2 Descriptors

Descriptors solve a common problem regarding management of variables.

To understand the issue, consider a `Car` class, that simulates a car.

Suppose that this class defines the variables `miles` and `kms`, which give the distance traveled in miles and kilometers respectively.

A highly simplified version of the class might look as follows

```
In [86]: class Car:

             def __init__(self, miles=1000):
                 self.miles = miles
                 self.kms = miles * 1.61

             # Some other functionality, details omitted
```

One potential problem we might have here is that a user alters one of these variables but not the other

```
In [87]: car = Car()
         car.miles

Out[87]: 1000

In [88]: car.kms

Out[88]: 1610.0

In [89]: car.miles = 6000
         car.kms
```

```
Out[89]: 1610.0
```

In the last two lines we see that `miles` and `kms` are out of sync.

What we really want is some mechanism whereby each time a user sets one of these variables, *the other is automatically updated.*

A Solution

In Python, this issue is solved using *descriptors*.

A descriptor is just a Python object that implements certain methods.

These methods are triggered when the object is accessed through dotted attribute notation.

The best way to understand this is to see it in action.

Consider this alternative version of the `Car` class

```
In [90]: class Car:

             def __init__(self, miles=1000):
                 self._miles = miles
                 self._kms = miles * 1.61

             def set_miles(self, value):
                 self._miles = value
                 self._kms = value * 1.61

             def set_kms(self, value):
                 self._kms = value
                 self._miles = value / 1.61

             def get_miles(self):
                 return self._miles

             def get_kms(self):
                 return self._kms

             miles = property(get_miles, set_miles)
             kms = property(get_kms, set_kms)
```

First let's check that we get the desired behavior

```
In [91]: car = Car()
         car.miles

Out[91]: 1000

In [92]: car.miles = 6000
         car.kms

Out[92]: 9660.0
```

Yep, that's what we want — `car.kms` is automatically updated.

How it Works

The names `_miles` and `_kms` are arbitrary names we are using to store the values of the variables.

The objects `miles` and `kms` are *properties*, a common kind of descriptor.

The methods `get_miles`, `set_miles`, `get_kms` and `set_kms` define what happens when you get (i.e. access) or set (bind) these variables

- So-called "getter" and "setter" methods.

The builtin Python function `property` takes getter and setter methods and creates a property.

For example, after `car` is created as an instance of `Car`, the object `car.miles` is a property.

Being a property, when we set its value via `car.miles = 6000` its setter method is triggered — in this case `set_miles`.

Decorators and Properties

These days its very common to see the `property` function used via a decorator.

Here's another version of our `Car` class that works as before but now uses decorators to set up the properties

```
In [93]: class Car:

             def __init__(self, miles=1000):
                 self._miles = miles
                 self._kms = miles * 1.61

             @property
             def miles(self):
                 return self._miles

             @property
             def kms(self):
                 return self._kms

             @miles.setter
             def miles(self, value):
                 self._miles = value
                 self._kms = value * 1.61

             @kms.setter
             def kms(self, value):
                 self._kms = value
                 self._miles = value / 1.61
```

We won't go through all the details here.

For further information you can refer to the descriptor documentation.

16.7 Generators

A generator is a kind of iterator (i.e., it works with a `next` function).

We will study two ways to build generators: generator expressions and generator functions.

16.7.1 Generator Expressions

The easiest way to build generators is using *generator expressions*.

Just like a list comprehension, but with round brackets.

Here is the list comprehension:

```
In [94]: singular = ('dog', 'cat', 'bird')
         type(singular)

Out[94]: tuple

In [95]: plural = [string + 's' for string in singular]
         plural

Out[95]: ['dogs', 'cats', 'birds']

In [96]: type(plural)

Out[96]: list
```

And here is the generator expression

```
In [97]: singular = ('dog', 'cat', 'bird')
         plural = (string + 's' for string in singular)
         type(plural)

Out[97]: generator

In [98]: next(plural)

Out[98]: 'dogs'

In [99]: next(plural)

Out[99]: 'cats'

In [100]: next(plural)

Out[100]: 'birds'
```

Since `sum()` can be called on iterators, we can do this

```
In [101]: sum((x * x for x in range(10)))
```

```
Out[101]: 285
```

The function `sum()` calls `next()` to get the items, adds successive terms.

In fact, we can omit the outer brackets in this case

```
In [102]: sum(x * x for x in range(10))
```

```
Out[102]: 285
```

16.7.2 Generator Functions

The most flexible way to create generator objects is to use generator functions.

Let's look at some examples.

Example 1

Here's a very simple example of a generator function

```
In [103]: def f():
              yield 'start'
              yield 'middle'
              yield 'end'
```

It looks like a function, but uses a keyword `yield` that we haven't met before.

Let's see how it works after running this code

```
In [104]: type(f)
```

```
Out[104]: function
```

```
In [105]: gen = f()
          gen
```

```
Out[105]: <generator object f at 0x7fb43d114570>
```

```
In [106]: next(gen)
```

```
Out[106]: 'start'
```

```
In [107]: next(gen)
```

```
Out[107]: 'middle'
```

```
In [108]: next(gen)
```

```
Out[108]: 'end'
```

```
In [109]: next(gen)
```

```
                                                                          
---------------------------------------------------------------------------

StopIteration                             Traceback (most recent call last)

<ipython-input-109-6e72e47198db> in <module>
----> 1 next(gen)


StopIteration:
```

The generator function `f()` is used to create generator objects (in this case `gen`).

Generators are iterators, because they support a `next` method.

The first call to `next(gen)`

- Executes code in the body of `f()` until it meets a `yield` statement.
- Returns that value to the caller of `next(gen)`.

The second call to `next(gen)` starts executing *from the next line*

```
In [110]: def f():
              yield 'start'
              yield 'middle'  # This line!
              yield 'end'
```

and continues until the next `yield` statement.

At that point it returns the value following `yield` to the caller of `next(gen)`, and so on.

When the code block ends, the generator throws a `StopIteration` error.

Example 2

Our next example receives an argument `x` from the caller

```
In [111]: def g(x):
              while x < 100:
                  yield x
                  x = x * x
```

Let's see how it works

```
In [112]: g
```

```
Out[112]: <function __main__.g(x)>

In [113]: gen = g(2)
          type(gen)

Out[113]: generator

In [114]: next(gen)

Out[114]: 2

In [115]: next(gen)

Out[115]: 4

In [116]: next(gen)

Out[116]: 16

In [117]: next(gen)
```

```
---------------------------------------------------------------------------

StopIteration                             Traceback (most recent call last)

<ipython-input-117-6e72e47198db> in <module>
----> 1 next(gen)


StopIteration:
```

The call `gen = g(2)` binds `gen` to a generator.

Inside the generator, the name `x` is bound to `2`.

When we call `next(gen)`

- The body of `g()` executes until the line `yield x`, and the value of `x` is returned.

Note that value of `x` is retained inside the generator.

When we call `next(gen)` again, execution continues *from where it left off*

```
In [ ]: def g(x):
            while x < 100:
                yield x
                x = x * x  # execution continues from here
```

When `x < 100` fails, the generator throws a `StopIteration` error.

Incidentally, the loop inside the generator can be infinite

```
In [118]: def g(x):
              while 1:
                  yield x
                  x = x * x
```

16.7.3 Advantages of Iterators

What's the advantage of using an iterator here?

Suppose we want to sample a binomial(n,0.5).

One way to do it is as follows

```
In [119]: import random
          n = 10000000
          draws = [random.uniform(0, 1) < 0.5 for i in range(n)]
          sum(draws)

Out[119]: 5000191
```

But we are creating two huge lists here, `range(n)` and `draws`.

This uses lots of memory and is very slow.

If we make `n` even bigger then this happens

```
In [120]: n = 100000000
          draws = [random.uniform(0, 1) < 0.5 for i in range(n)]
```

We can avoid these problems using iterators.

Here is the generator function

```
In [121]: def f(n):
              i = 1
              while i <= n:
                  yield random.uniform(0, 1) < 0.5
                  i += 1
```

Now let's do the sum

```
In [122]: n = 10000000
          draws = f(n)
          draws

Out[122]: <generator object f at 0x7fb44c3848b8>

In [123]: sum(draws)
```

```
Out[123]: 5000804
```

In summary, iterables

- avoid the need to create big lists/tuples, and
- provide a uniform interface to iteration that can be used transparently in `for` loops

16.8 Recursive Function Calls

This is not something that you will use every day, but it is still useful — you should learn it at some stage.

Basically, a recursive function is a function that calls itself.

For example, consider the problem of computing x_t for some t when

$$x_{t+1} = 2x_t, \quad x_0 = 1 \tag{1}$$

Obviously the answer is 2^t.

We can compute this easily enough with a loop

```
In [124]: def x_loop(t):
              x = 1
              for i in range(t):
                  x = 2 * x
              return x
```

We can also use a recursive solution, as follows

```
In [125]: def x(t):
              if t == 0:
                  return 1
              else:
                  return 2 * x(t-1)
```

What happens here is that each successive call uses it's own *frame* in the *stack*

- a frame is where the local variables of a given function call are held
- stack is memory used to process function calls
  - a First In Last Out (FILO) queue

This example is somewhat contrived, since the first (iterative) solution would usually be preferred to the recursive solution.

We'll meet less contrived applications of recursion later on.

16.9 Exercises

16.9.1 Exercise 1

The Fibonacci numbers are defined by

$$x_{t+1} = x_t + x_{t-1}, \quad x_0 = 0,\ x_1 = 1 \tag{2}$$

The first few numbers in the sequence are $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55$.

Write a function to recursively compute the t-th Fibonacci number for any t.

16.9.2 Exercise 2

Complete the following code, and test it using this csv file, which we assume that you've put in your current working directory

```
def column_iterator(target_file, column_number):
    """A generator function for CSV files.
    When called with a file name target_file (string) and column number
    column_number (integer), the generator function returns a generator
    that steps through the elements of column column_number in file
    target_file.
    """
    # put your code here

dates = column_iterator('test_table.csv', 1)

for date in dates:
    print(date)
```

16.9.3 Exercise 3

Suppose we have a text file `numbers.txt` containing the following lines

```
prices
3
8

7
21
```

Using `try` – `except`, write a program to read in the contents of the file and sum the numbers, ignoring lines without numbers.

16.10 Solutions

16.10.1 Exercise 1

Here's the standard solution

```
In [126]: def x(t):
              if t == 0:
```

```
            return 0
        if t == 1:
            return 1
        else:
            return x(t-1) + x(t-2)
```

Let's test it

```
In [127]: print([x(i) for i in range(10)])
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

16.10.2 Exercise 2

One solution is as follows

```
In [128]: def column_iterator(target_file, column_number):
              """A generator function for CSV files.
              When called with a file name target_file (string) and column number
              column_number (integer), the generator function returns a generator
              which steps through the elements of column column_number in file
              target_file.
              """
              f = open(target_file, 'r')
              for line in f:
                  yield line.split(',')[column_number - 1]
              f.close()

          dates = column_iterator('test_table.csv', 1)

          i = 1
          for date in dates:
              print(date)
              if i == 10:
                  break
              i += 1
```

```
    Date
2009-05-21
2009-05-20
2009-05-19
2009-05-18
2009-05-15
2009-05-14
2009-05-13
2009-05-12
2009-05-11
```

16.10.3 Exercise 3

Let's save the data first

In [129]:
```
%%file numbers.txt
prices
3
8

7
21
```

```
Overwriting numbers.txt
```

In [130]:
```
f = open('numbers.txt')

total = 0.0
for line in f:
    try:
        total += float(line)
    except ValueError:
        pass

f.close()

print(total)
```

```
39.0
```

Chapter 17

Debugging

17.1 Contents

- Overview 17.2
- Debugging 17.3
- Other Useful Magics 17.4

> "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." – Brian Kernighan

17.2 Overview

Are you one of those programmers who fills their code with `print` statements when trying to debug their programs?

Hey, we all used to do that.

(OK, sometimes we still do that…)

But once you start writing larger programs you'll need a better system.

Debugging tools for Python vary across platforms, IDEs and editors.

Here we'll focus on Jupyter and leave you to explore other settings.

We'll need the following imports

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

17.3 Debugging

17.3.1 The `debug` Magic

Let's consider a simple (and rather contrived) example

```
In [2]: def plot_log():
            fig, ax = plt.subplots(2, 1)
            x = np.linspace(1, 2, 10)
            ax.plot(x, np.log(x))
            plt.show()

        plot_log()  # Call the function, generate plot
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-2-c32a2280f47b> in <module>
      5     plt.show()
      6
----> 7 plot_log()  # Call the function, generate plot


<ipython-input-2-c32a2280f47b> in plot_log()
      2     fig, ax = plt.subplots(2, 1)
      3     x = np.linspace(1, 2, 10)
----> 4     ax.plot(x, np.log(x))
      5     plt.show()
      6


AttributeError: 'numpy.ndarray' object has no attribute 'plot'
```

This code is intended to plot the `log` function over the interval $[1, 2]$.

But there's an error here: `plt.subplots(2, 1)` should be just `plt.subplots()`.

(The call `plt.subplots(2, 1)` returns a NumPy array containing two axes objects, suitable for having two subplots on the same figure)

The traceback shows that the error occurs at the method call `ax.plot(x, np.log(x))`.

The error occurs because we have mistakenly made `ax` a NumPy array, and a NumPy array has no `plot` method.

But let's pretend that we don't understand this for the moment.

We might suspect there's something wrong with `ax` but when we try to investigate this object, we get the following exception:

```
In [3]: ax
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-b00e77935981> in <module>
----> 1 ax

NameError: name 'ax' is not defined
```

The problem is that `ax` was defined inside `plot_log()`, and the name is lost once that function terminates.

Let's try doing it a different way.

We run the first cell block again, generating the same error

```
In [4]: def plot_log():
            fig, ax = plt.subplots(2, 1)
            x = np.linspace(1, 2, 10)
            ax.plot(x, np.log(x))
            plt.show()

        plot_log()  # Call the function, generate plot
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
```

```
    <ipython-input-4-c32a2280f47b> in <module>
      5     plt.show()
      6
----> 7 plot_log()  # Call the function, generate plot


    <ipython-input-4-c32a2280f47b> in plot_log()
      2     fig, ax = plt.subplots(2, 1)
      3     x = np.linspace(1, 2, 10)
----> 4     ax.plot(x, np.log(x))
      5     plt.show()
      6


   AttributeError: 'numpy.ndarray' object has no attribute 'plot'
```

But this time we type in the following cell block

```
%debug
```

You should be dropped into a new prompt that looks something like this

```
ipdb>
```

(You might see pdb> instead)

Now we can investigate the value of our variables at this point in the program, step forward through the code, etc.

For example, here we simply type the name `ax` to see what's happening with this object:

```
ipdb> ax
array([<matplotlib.axes.AxesSubplot object at 0x290f5d0>,
       <matplotlib.axes.AxesSubplot object at 0x2930810>], dtype=object)
```

It's now very clear that `ax` is an array, which clarifies the source of the problem.

To find out what else you can do from inside `ipdb` (or `pdb`), use the online help

```
ipdb> h

Documented commands (type help <topic>):
========================================
EOF    bt         cont      enable  jump  pdef   r        tbreak   w
a      c          continue  exit    l     pdoc   restart  u        whatis
alias  cl         d         h       list  pinfo  return   unalias  where
args   clear      debug     help    n     pp     run      unt
b      commands   disable   ignore  next  q      s        until
break  condition  down      j       p     quit   step     up

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
retval  rv

ipdb> h c
c(ont(inue))
Continue execution, only stop when a breakpoint is encountered.
```

17.3.2 Setting a Break Point

The preceding approach is handy but sometimes insufficient.

Consider the following modified version of our function above

```
In [5]: def plot_log():
            fig, ax = plt.subplots()
            x = np.logspace(1, 2, 10)
            ax.plot(x, np.log(x))
            plt.show()

        plot_log()
```

Here the original problem is fixed, but we've accidentally written `np.logspace(1, 2, 10)` instead of `np.linspace(1, 2, 10)`.

Now there won't be any exception, but the plot won't look right.

To investigate, it would be helpful if we could inspect variables like `x` during execution of the function.

To this end, we add a "break point" by inserting `breakpoint()` inside the function code block

```
def plot_log():
    breakpoint()
    fig, ax = plt.subplots()
    x = np.logspace(1, 2, 10)
    ax.plot(x, np.log(x))
    plt.show()

plot_log()
```

Now let's run the script, and investigate via the debugger

```
> <ipython-input-6-a188074383b7>(6)plot_log()
-> fig, ax = plt.subplots()
(Pdb) n
> <ipython-input-6-a188074383b7>(7)plot_log()
-> x = np.logspace(1, 2, 10)
(Pdb) n
> <ipython-input-6-a188074383b7>(8)plot_log()
-> ax.plot(x, np.log(x))
(Pdb) x
```

```
array([ 10.        ,  12.91549665,  16.68100537,  21.5443469 ,
        27.82559402,  35.93813664,  46.41588834,  59.94842503,
        77.42636827, 100.        ])
```

We used `n` twice to step forward through the code (one line at a time).

Then we printed the value of `x` to see what was happening with that variable.

To exit from the debugger, use `q`.

17.4 Other Useful Magics

In this lecture, we used the `%debug` IPython magic.

There are many other useful magics:

- `%precision 4` sets printed precision for floats to 4 decimal places
- `%whos` gives a list of variables and their values
- `%quickref` gives a list of magics

The full list of magics is here.

Part IV

Tools and Techniques

Chapter 18

Geometric Series for Elementary Economics

18.1 Contents

- Overview 18.2
- Key Formulas 18.3
- Example: The Money Multiplier in Fractional Reserve Banking 18.4
- Example: The Keynesian Multiplier 18.5
- Example: Interest Rates and Present Values 18.6
- Back to the Keynesian Multiplier 18.7

18.2 Overview

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

Among these are

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

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

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

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

Below we'll use the following imports:

In [1]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import sympy as sym
from sympy import init_printing
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
```

18.3 Key Formulas

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

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

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

18.3.1 Infinite Geometric Series

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

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

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

The key formula is

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

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

18.3.2 Finite Geometric Series

The second series that interests us is the finite geomtric series

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

where T is a positive integer.

The key formula here is

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

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

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

18.4 Example: The Money Multiplier in Fractional Reserve Banking

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

- In recent times
  - cash consists of pieces of paper issued by the government and called dollars or pounds or ...
  - a *deposit* is a balance in a checking or savings account that entitles the owner to ask the bank for immediate payment in cash
- When the UK and France and the US were on either a gold or silver standard (before 1914, for example)
  - cash was a gold or silver coin
  - a *deposit receipt* was a *bank note* that the bank promised to convert into gold or silver on demand; (sometimes it was also a checking or savings account balance)

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

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

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

The geometric series formula (1) is at the heart of the classic model of the money creation process – one that leads us to the celebrated **money multiplier**.

18.4.1 A Simple Model

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

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

$$L_i + R_i = D_i$$

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

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

Each bank i sets its reserves to satisfy the equation

$$R_i = rD_i \tag{2}$$

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

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

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

$$D_{i+1} = L_i \tag{3}$$

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

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

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

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

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

Now we do a little algebra.

Combining equations (2) and (3) tells us that

$$L_i = (1-r)D_i \tag{4}$$

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

Combining equation (4) with equation (3) tells us that

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

which implies that

$$D_i = (1-r)^i D_0 \quad \text{for } i \geq 0 \tag{5}$$

Equation (5) expresses D_i as the i th term in the product of D_0 and the geometric series

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

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

$$\sum_{i=0}^{\infty} (1-r)^i D_0 = \frac{D_0}{1-(1-r)} = \frac{D_0}{r} \tag{6}$$

18.4.2 Money Multiplier

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

Equation (6) asserts that the **money multiplier** is $\frac{1}{r}$

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

18.5 Example: The Keynesian Multiplier

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

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

18.5.1 Static Version

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

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

$$c + i = y$$

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

$$c = by$$

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

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

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

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

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

Solving this equation for y gives

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

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

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

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

where t is a nonnegative integer.

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

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

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

18.5.2 Dynamic Version

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

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

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

We modify our consumption function to assume the form

$$c_t = by_{t-1}$$

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

We begin wtih an initial condition stating that

$$y_{-1} = 0$$

We also assume that

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

so that investment is constant over time.

It follows that

$$y_0 = i + c_0 = i + by_{-1} = i$$

and

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

and

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

and more generally

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

or

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

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

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

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

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

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

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

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

$$c_t + i_t + g_t = y_t$$

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

18.6 Example: Interest Rates and Present Values

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

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

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

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

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

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

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

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

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

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

- If I sell x dollars to you today, you pay me Rx dollars tomorrow.

- This means that you borrowed x dollars for me at a gross interest rate R and a net interest rate r.

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

Two important geometric sequences are

$$1, R, R^2, \cdots \tag{7}$$

and

$$1, R^{-1}, R^{-2}, \cdots \tag{8}$$

Sequence (7) tells us how dollar values of an investment **accumulate** through time.

Sequence (8) tells us how to **discount** future dollars to get their values in terms of today's dollars.

18.6.1 Accumulation

Geometric sequence (7) tells us how one dollar invested and re-invested in a project with gross one period nominal rate of return accumulates

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

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

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

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

18.6.2 Discounting

Geometric sequence (8) tells us how much future dollars are worth in terms of today's dollars.

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

It follows that

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

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

18.6.3 Application to Asset Pricing

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

$$x_t = G^t x_0$$

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

Thus, lease payments increase at g percent per period.

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

The **present value** of the lease is

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

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

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

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

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

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

Use this approximation to write p_0 as

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

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

The approximation

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

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

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

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

$$x_t = G^t x_0$$

The present value of this lease is:

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

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

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

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

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

Thus, we get the following approximation:

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

Expanding:

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

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

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

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

In [2]:
```
# True present value of a finite lease
def finite_lease_pv_true(T, g, r, x_0):
    G = (1 + g)
    R = (1 + r)
    return (x_0 * (1 - G**(T + 1) * R**(-T - 1))) / (1 - G * R**(-1))
# First approximation for our finite lease

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

```
    return p

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

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

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

First we study the quality of our approximations

In [3]:
```
def plot_function(axes, x_vals, func, args):
    axes.plot(x_vals, func(*args), label=func.__name__)

T_max = 50

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

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

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

Evidently our approximations perform well for small values of T.

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

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

```
In [4]: # Convergence of infinite and finite
        T_max = 1000
        T = np.arange(0, T_max+1)
        fig, ax = plt.subplots()
        ax.set_title('Infinite and Finite Lease Present Value $T$ Periods Ahead')
        f_1 = finite_lease_pv_true(T, g, r, x_0)
        f_2 = np.ones(T_max+1)*infinite_lease(g, r, x_0)
        ax.plot(T, f_1, label='T-period lease PV')
        ax.plot(T, f_2, '--', label='Infinite lease PV')
        ax.set_xlabel('$T$ Periods Ahead')
        ax.set_ylabel('Present Value, $p_0$')
        ax.legend()
        plt.show()
```

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

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

```
In [5]: # First view
        # Changing r and g
        fig, ax = plt.subplots()
        ax.set_title('Value of lease of length $T$')
        ax.set_ylabel('Present Value, $p_0$')
        ax.set_xlabel('$T$ periods ahead')
        T_max = 10
        T=np.arange(0, T_max+1)

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

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

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

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

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

In [6]:
```
# Second view
fig = plt.figure()
T = 3
ax = fig.gca(projection='3d')
r = np.arange(0.01, 0.99, 0.005)
g = np.arange(0.011, 0.991, 0.005)

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

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

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

We will use a library called SymPy.

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

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

After that, we'll use SymPy to compute derivatives

In [7]:
```
# Creates algebraic symbols that can be used in an algebraic expression
g, r, x0 = sym.symbols('g, r, x0')
G = (1 + g)
R = (1 + r)
p0 = x0 / (1 - G * R**(-1))
init_printing()
print('Our formula is:')
p0
```

```
Our formula is:
```

Out[7]: $$\frac{x_0}{-\frac{g+1}{r+1} + 1}$$

In [8]:
```
print('dp0 / dg is:')
dp_dg = sym.diff(p0, g)
dp_dg
```

```
dp0 / dg is:
```

Out[8]: $$\frac{x_0}{(r+1)\left(-\frac{g+1}{r+1}+1\right)^2}$$

In [9]:
```
print('dp0 / dr is:')
dp_dr = sym.diff(p0, r)
dp_dr
```

```
dp0 / dr is:
```

Out[9]: $$-\frac{x_0\,(g+1)}{(r+1)^2\left(-\frac{g+1}{r+1}+1\right)^2}$$

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

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

18.7 Back to the Keynesian Multiplier

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

In [10]:
```
# Function that calculates a path of y
def calculate_y(i, b, g, T, y_init):
    y = np.zeros(T+1)
    y[0] = i + b * y_init + g
    for t in range(1, T+1):
        y[t] = b * y[t-1] + i + g
    return y

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

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

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

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

```
In [11]: bs = (1/3, 2/3, 5/6, 0.9)

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

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

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

```
In [12]: fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 10))
         fig.subplots_adjust(hspace=0.3)

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

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

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

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

Chapter 19

Linear Algebra

19.1 Contents

- Overview 19.2
- Vectors 19.3
- Matrices 19.4
- Solving Systems of Equations 19.5
- Eigenvalues and Eigenvectors 19.6
- Further Topics 19.7
- Exercises 19.8
- Solutions 19.9

19.2 Overview

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

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

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

or, more generally,

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

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

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

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

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

These are the kinds of topics addressed by linear algebra.

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

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

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

Let's start with some imports:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
from scipy.interpolate import interp2d
from scipy.linalg import inv, solve, det, eig
```

19.3 Vectors

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

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

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

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

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

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

The following figure represents three vectors in this manner

In [2]:
```
fig, ax = plt.subplots(figsize=(10, 8))
# Set the axes through the origin
for spine in ['left', 'bottom']:
    ax.spines[spine].set_position('zero')
for spine in ['right', 'top']:
    ax.spines[spine].set_color('none')

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

19.3.1 Vector Operations

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

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

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

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

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

Scalar multiplication is illustrated in the next figure

```
In [3]: fig, ax = plt.subplots(figsize=(10, 8))
        # Set the axes through the origin
        for spine in ['left', 'bottom']:
```

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

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


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

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

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

more commonly represented as a NumPy array.

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

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

```
Out[4]: array([3., 5., 7.])
```

```
In [5]: 4 * x
```

```
Out[5]: array([4., 4., 4.])
```

19.3.2 Inner Product and Norm

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

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

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

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

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

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

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

```
In [6]: np.sum(x * y)          # Inner product of x and y
```

```
Out[6]: 12.0
```

```
In [7]: np.sqrt(np.sum(x**2))  # Norm of x, take one
```

```
Out[7]: 1.7320508075688772
```

```
In [8]: np.linalg.norm(x)      # Norm of x, take two
```

```
Out[8]: 1.7320508075688772
```

19.3.3 Span

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

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

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

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

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

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

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

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

In [9]:
```
fig = plt.figure(figsize=(10, 8))
ax = fig.gca(projection='3d')

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

α, β = 0.2, 0.1

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

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


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

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

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

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

Examples

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

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

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

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

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

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

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

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

19.3.4 Linear Independence

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

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

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

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

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

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

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

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

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

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

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

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

19.3.5 Unique Representations

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

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

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

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

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

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

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

19.4 Matrices

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

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

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

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

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

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

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

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

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

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

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

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

19.4.1 Matrix Operations

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

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

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

and

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

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

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

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

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

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

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

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

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

$$Ax = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} := \begin{bmatrix} a_{11}x_1 + \cdots + a_{1k}x_k \\ \vdots \\ a_{n1}x_1 + \cdots + a_{nk}x_k \end{bmatrix} \tag{2}$$

Note

AB and BA are not generally the same thing.

Another important special case is the identity matrix.

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

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

19.4.2 Matrices in NumPy

NumPy arrays are also used as matrices, and have fast, efficient functions and methods for all the standard matrix operations Section **??**.

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

```
In [10]: A = ((1, 2),
              (3, 4))

         type(A)
```

```
Out[10]: tuple
```

```
In [11]: A = np.array(A)

         type(A)
```

```
Out[11]: numpy.ndarray
```

```
In [12]: A.shape
```

```
Out[12]: (2, 2)
```

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

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

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

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

```
In [13]: A = np.identity(3)
         B = np.ones((3, 3))
         2 * A
```

```
Out[13]: array([[2., 0., 0.],
                [0., 2., 0.],
                [0., 0., 2.]])
```

```
In [14]: A + B
```

```
Out[14]: array([[2., 1., 1.],
                [1., 2., 1.],
                [1., 1., 2.]])
```

To multiply matrices we use the `@` symbol.

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

See here for more discussion.

19.4.3 Matrices as Maps

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

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

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

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

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

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

19.5 Solving Systems of Equations

Recall again the system of equations (1).

If we compare (1) and (2), we see that (1) can now be written more conveniently as

$$y = Ax \tag{3}$$

The problem we face is to determine a vector $x \in \mathbb{R}^k$ that solves (3), taking y and A as given.

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

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

If so, is it always unique?

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

In [15]:
```
def f(x):
    return 0.6 * np.cos(4 * x) + 1.4


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

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

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

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

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

ax = axes[0]

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

ax = axes[1]

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

plt.show()
```

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

Can we impose conditions on A in (3) that rule out these problems?

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

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

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

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

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

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

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

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

19.5.1 The Square Matrix Case

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

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

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

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

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

Moreover, the solution is unique.

In particular, the following are equivalent

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

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

Inverse Matrices

Can we give some sort of expression for the solution?

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

A similar expression is available in the matrix case.

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

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

This is the solution that we're looking for.

Determinants

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

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

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

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

19.5.2 More Rows than Columns

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

19.5.3 More Columns than Rows

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

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

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

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

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

(For example, use the canonical basis vectors)

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

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

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

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

In other words, uniqueness fails.

19.5.4 Linear Equations with SciPy

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

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

```
In [16]: A = ((1, 2), (3, 4))
         A = np.array(A)
         y = np.ones((2, 1))  # Column vector
         det(A)  # Check that A is nonsingular, and hence invertible
```

```
Out[16]: -2.0
```

```
In [17]: A_inv = inv(A)  # Compute the inverse
         A_inv
```

```
Out[17]: array([[-2. ,  1. ],
                [ 1.5, -0.5]])
```

```
In [18]: x = A_inv @ y  # Solution
         A @ x          # Should equal y
```

```
Out[18]: array([[1.],
                [1.]])
```

```
In [19]: solve(A, y)  # Produces the same solution
```

```
Out[19]: array([[-1.],
                [ 1.]])
```

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

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

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

19.6 Eigenvalues and Eigenvectors

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

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

$$Av = \lambda v$$

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

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

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

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

In [20]:
```
A = ((1, 2),
     (2, 1))
A = np.array(A)
evals, evecs = eig(A)
evecs = evecs[:, 0], evecs[:, 1]

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

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

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

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

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

plt.show()
```

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

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

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

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

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

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

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

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

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

```
In [21]: A = ((1, 2),
              (2, 1))

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

```
Out[21]: array([ 3.+0.j, -1.+0.j])
```

```
In [22]: evecs
```

```
Out[22]: array([[ 0.70710678, -0.70710678],
                [ 0.70710678,  0.70710678]])
```

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

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

19.6.1 Generalized Eigenvalues

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

$$Av = \lambda Bv$$

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

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

19.7 Further Topics

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

19.7.1 Series Expansions

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

A generalization of this idea exists in the matrix setting.

Matrix Norms

Let A be a square matrix, and let

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

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

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

Neumann's Theorem

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

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

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

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k \tag{4}$$

Spectral Radius

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

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

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

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

In which case (4) is valid.

19.7.2 Positive Definite Matrices

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

We say that A is

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

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

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

19.7.3 Differentiating Linear and Quadratic Forms

The following formulas are useful in many economic contexts. Let

- z, x and a all be $n \times 1$ vectors

- A be an $n \times n$ matrix
- B be an $m \times n$ matrix and y be an $m \times 1$ vector

Then

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

Exercise 1 below asks you to apply these formulas.

19.7.4 Further Reading

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

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

If you don't mind a slightly abstract approach, a nice intermediate-level text on linear algebra is [89].

19.8 Exercises

19.8.1 Exercise 1

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

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

subject to the linear constraint

$$y = Ax + Bu$$

Here

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

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

One way to solve the problem is to form the Lagrangian

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

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

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

Show that these conditions imply that

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

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

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

19.9 Solutions

19.9.1 Solution to Exercise 1

We have an optimization problem:

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

s.t.

$$y = Ax + Bu$$

with primitives

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

The associated Lagrangian is:

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

1. $\widehat{\ }$.

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

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

since P is symmetric.

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

$$\lambda = -2Py$$

2. ^^.

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

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

Substituting $\lambda = -2Py$ gives

$$Qu + B'Py = 0$$

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

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

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

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

Thus, the optimal choice of u must satisfy

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

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

3. ^^.

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

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

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

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

To evaluate the function

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

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

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

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

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

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

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

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

This implies that

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

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

Footnotes

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

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

Chapter 20

Complex Numbers and Trigonometry

20.1 Contents

- Overview 20.2
- De Moivre's Theorem 20.3
- Applications of de Moivre's Theorem 20.4

20.2 Overview

This lecture introduces some elementary mathematics and trigonometry.

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

For example, these tools are keys to understanding outcomes attained by Paul Samuelson (1939) [139] in his classic paper on interactions between the investment accelerator and the Keynesian consumption function, our topic in the lecture Samuelson Multiplier Accelerator.

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

So let's dive in.

20.2.1 Complex Numbers

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

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

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

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

- Euler contributed many other formulas too!

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

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

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

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

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

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

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

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

Therefore,

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

Three elementary trigonometric functions are

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

We'll need the following imports:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sympy import *
```

20.2.2 An Example

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

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

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

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

In [2]:
```
# Abbreviate useful values and functions
π = np.pi
zeros = np.zeros
ones = np.ones

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

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

ax.plot((0, θ), (0, r), marker='o', color='b')          # Plot r
ax.plot(zeros(x_range.shape), x_range, color='b')       # Plot x
ax.plot(θ_range, x / np.cos(θ_range), color='b')        # Plot y
ax.plot(θ_range, ones(θ_range.shape) * 0.1, color='r')  # Plot ϑ

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

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

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

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

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

20.3 De Moivre's Theorem

de Moivre's theorem states that:

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

To prove de Moivre's theorem, note that

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

and compute.

20.4 Applications of de Moivre's Theorem

20.4.1 Example 1

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

We have

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

and thus

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

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

20.4.2 Example 2

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

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

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

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

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

Thus,

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

20.4.3 Example 3

This example provides machinery that is at the heard of Samuelson's analysis of his multiplier-accelerator model [139].

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

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

whose **characteristic polynomial** is

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

or

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

has roots z_1, z_1.

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

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

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

To solve the difference equation, recall from example 2 that

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

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

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

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

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

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

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

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

In [3]:
```
# Set parameters
r = 0.9
θ = π/4
x0 = 4
x1 = 2 * r * sqrt(2)

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

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

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

```
    ω = 0.000
p = 2.000
```

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

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

In [4]:
```
# Define range of n
max_n = 30
n = np.arange(0, max_n+1, 0.01)

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

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

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

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

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

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

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

20.4.4 Trigonometric Identities

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

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

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

For example, we'll calculate identities for

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

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

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

We can also obtain the trigonometric identities as follows:

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

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

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

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

In [5]:
```
# Define symbols
ω, θ = symbols('ω θ', real=True)

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

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

20.4.5 Trigonometric Integrals

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

For example, we want to solve the following integral:

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

Using Euler's formula, we have:

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

and thus:

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

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

In [6]:
```
# Set initial printing
init_printing()

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

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

Out[6]: $\frac{\sin^2(\omega)}{2}$

In [7]:
```
print('The numerical solution for the integral of cos(ω)sin(ω) \
from -π to π is:')
integrate(cos(ω) * sin(ω), (ω, -π, π))
```

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

Out[7]: 0

20.4.6 Exercises

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

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

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

Chapter 21

Orthogonal Projections and Their Applications

21.1 Contents

- Overview 21.2
- Key Definitions 21.3
- The Orthogonal Projection Theorem 21.4
- Orthonormal Basis 21.5
- Projection Using Matrix Algebra 21.6
- Least Squares Regression 21.7
- Orthogonalization and Decomposition 21.8
- Exercises 21.9
- Solutions 21.10

21.2 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:

```
In [1]: import numpy as np
        from scipy.linalg import qr
```

21.2.1 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, [132].

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

21.3 Key Definitions

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

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

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

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

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

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

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

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

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

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

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

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

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

21.3.1 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.

21.4 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 provides answer to 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

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

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

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

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

The next figure provides some intuition

21.4.1 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.

21.4.2 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.

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

21.5 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 $\operatorname{span} O = S$, 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 $x \in \operatorname{span}\{u_1, \dots, u_k\}$, we can find scalars $\alpha_1, \dots, \alpha_k$ that verify

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

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) verifies the claim.

21.5.1 Projection onto an Orthonormal Basis

When the subspace onto which are projecting is orthonormal, computing the projection simplifies:

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

$$Py = \sum_{i=1}^{k} \langle y, u_i \rangle u_i, \quad \forall\, y \in \mathbb{R}^n \tag{2}$$

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

This is true because

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

21.6 Projection Using 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 https://en.wikipedia.org/wiki/Linear_map#Matrices.

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

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

This is true: 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.

21.6.1 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**.

21.6.2 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.

21.6.3 Application: Overdetermined Systems of Equations

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

Given X and y, we seek $b \in \mathbb{R}^k$ satisfying 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 as close to y as possible.

The next theorem shows that the solution 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.

21.7 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.

21.7.1 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 $1/N$, 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**.

21.7.2 Solution

Define the matrices

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

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

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

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.

21.8 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.

21.8.1 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

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

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.

21.8.2 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 $X = QR$.

21.8.3 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.

21.9 Exercises

21.9.1 Exercise 1

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

21.9.2 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?

21.9.3 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

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

21.10 Solutions

21.10.1 Exercise 1

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

21.10.2 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.).

21.10.3 Exercise 3

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

In [2]:
```
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

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

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

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

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

In [4]:
```
Py1 = X @ np.linalg.inv(X.T @ X) @ X.T @ y
Py1
```

Out[4]:
```
array([-0.56521739,  3.26086957, -2.2173913 ])
```

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

```
In [5]: U = gram_schmidt(X)
        U
```

```
Out[5]: array([[ 0.4472136 , -0.13187609],
               [ 0.        , -0.98907071],
               [ 0.89442719,  0.06593805]])
```

```
In [6]: Py2 = U @ U.T @ y
        Py2
```

```
Out[6]: 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:

```
In [7]: Q, R = qr(X, mode='economic')
        Q
```

```
Out[7]: array([[-0.4472136 , -0.13187609],
               [-0.        , -0.98907071],
               [-0.89442719,  0.06593805]])
```

```
In [8]: Py3 = Q @ Q.T @ y
        Py3
```

```
Out[8]: array([-0.56521739,  3.26086957, -2.2173913 ])
```

Again, we obtain the same answer.

Chapter 22

LLN and CLT

22.1 Contents

- Overview 22.2
- Relationships 22.3
- LLN 22.4
- CLT 22.5
- Exercises 22.6
- Solutions 22.7

22.2 Overview

This lecture illustrates two of the most important theorems of probability and statistics: The law of large numbers (LLN) and the central limit theorem (CLT).

These beautiful theorems lie behind many of the most fundamental results in econometrics and quantitative economic modeling.

The lecture is based around simulations that show the LLN and CLT in action.

We also demonstrate how the LLN and CLT break down when the assumptions they are based on do not hold.

In addition, we examine several useful extensions of the classical theorems, such as

- The delta method, for smooth functions of random variables.
- The multivariate case.

Some of these extensions are presented as exercises.

We'll need the following imports:

In [1]:
```
import random
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.stats import t, beta, lognorm, expon, gamma, uniform, cauchy
from scipy.stats import gaussian_kde, poisson, binom, norm, chi2
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.collections import PolyCollection
from scipy.linalg import inv, sqrtm
```

22.3 Relationships

The CLT refines the LLN.

The LLN gives conditions under which sample moments converge to population moments as sample size increases.

The CLT provides information about the rate at which sample moments converge to population moments as sample size increases.

22.4 LLN

We begin with the law of large numbers, which tells us when sample averages will converge to their population means.

22.4.1 The Classical LLN

The classical law of large numbers concerns independent and identically distributed (IID) random variables.

Here is the strongest version of the classical LLN, known as *Kolmogorov's strong law*.

Let $X_1, \dots, X_n$ be independent and identically distributed scalar random variables, with common distribution F.

When it exists, let μ denote the common mean of this sample:

$$\mu := \mathbb{E}X = \int x F(dx)$$

In addition, let

$$\bar{X}_n := \frac{1}{n} \sum_{i=1}^{n} X_i$$

Kolmogorov's strong law states that, if $\mathbb{E}|X|$ is finite, then

$$\mathbb{P}\left\{\bar{X}_n \to \mu \text{ as } n \to \infty\right\} = 1 \tag{1}$$

What does this last expression mean?

Let's think about it from a simulation perspective, imagining for a moment that our computer can generate perfect random samples (which of course it can't).

Let's also imagine that we can generate infinite sequences so that the statement $\bar{X}_n \to \mu$ can be evaluated.

In this setting, (1) should be interpreted as meaning that the probability of the computer producing a sequence where $\bar{X}_n \to \mu$ fails to occur is zero.

22.4.2 Proof

The proof of Kolmogorov's strong law is nontrivial – see, for example, theorem 8.3.5 of [49].

On the other hand, we can prove a weaker version of the LLN very easily and still get most of the intuition.

The version we prove is as follows: If $X_1, \dots, X_n$ is IID with $\mathbb{E}X_i^2 < \infty$, then, for any $\epsilon > 0$, we have

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \to 0 \quad \text{as} \quad n \to \infty \tag{2}$$

(This version is weaker because we claim only convergence in probability rather than almost sure convergence, and assume a finite second moment)

To see that this is so, fix $\epsilon > 0$, and let σ^2 be the variance of each X_i.

Recall the Chebyshev inequality, which tells us that

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \leq \frac{\mathbb{E}[(\bar{X}_n - \mu)^2]}{\epsilon^2} \tag{3}$$

Now observe that

$$\begin{aligned}
\mathbb{E}[(\bar{X}_n - \mu)^2] &= \mathbb{E}\left\{\left[\frac{1}{n}\sum_{i=1}^n (X_i - \mu)\right]^2\right\} \\
&= \frac{1}{n^2}\sum_{i=1}^n \sum_{j=1}^n \mathbb{E}(X_i - \mu)(X_j - \mu) \\
&= \frac{1}{n^2}\sum_{i=1}^n \mathbb{E}(X_i - \mu)^2 \\
&= \frac{\sigma^2}{n}
\end{aligned}$$

Here the crucial step is at the third equality, which follows from independence.

Independence means that if $i \neq j$, then the covariance term $\mathbb{E}(X_i - \mu)(X_j - \mu)$ drops out.

As a result, $n^2 - n$ terms vanish, leading us to a final expression that goes to zero in n.

Combining our last result with (3), we come to the estimate

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \leq \frac{\sigma^2}{n\epsilon^2} \tag{4}$$

The claim in (2) is now clear.

Of course, if the sequence $X_1, \dots, X_n$ is correlated, then the cross-product terms $\mathbb{E}(X_i - \mu)(X_j - \mu)$ are not necessarily zero.

While this doesn't mean that the same line of argument is impossible, it does mean that if we want a similar result then the covariances should be "almost zero" for "most" of these terms.

In a long sequence, this would be true if, for example, $\mathbb{E}(X_i - \mu)(X_j - \mu)$ approached zero when the difference between i and j became large.

In other words, the LLN can still work if the sequence $X_1, \dots, X_n$ has a kind of "asymptotic independence", in the sense that correlation falls to zero as variables become further apart in the sequence.

This idea is very important in time series analysis, and we'll come across it again soon enough.

22.4.3 Illustration

Let's now illustrate the classical IID law of large numbers using simulation.

In particular, we aim to generate some sequences of IID random variables and plot the evolution of $\bar{X}_n$ as n increases.

Below is a figure that does just this (as usual, you can click on it to expand it).

It shows IID observations from three different distributions and plots $\bar{X}_n$ against n in each case.

The dots represent the underlying observations X_i for $i = 1, \dots, 100$.

In each of the three cases, convergence of $\bar{X}_n$ to μ occurs as predicted

In [2]:
```
n = 100

# Arbitrary collection of distributions
distributions = {"student's t with 10 degrees of freedom": t(10),
                 "β(2, 2)": beta(2, 2),
                 "lognormal LN(0, 1/2)": lognorm(0.5),
                 "γ(5, 1/2)": gamma(5, scale=2),
                 "poisson(4)": poisson(4),
                 "exponential with λ = 1": expon(1)}

# Create a figure and some axes
num_plots = 3
fig, axes = plt.subplots(num_plots, 1, figsize=(10, 20))

# Set some plotting parameters to improve layout
bbox = (0., 1.02, 1., .102)
legend_args = {'ncol': 2,
               'bbox_to_anchor': bbox,
               'loc': 3,
               'mode': 'expand'}
plt.subplots_adjust(hspace=0.5)

for ax in axes:
    # Choose a randomly selected distribution
    name = random.choice(list(distributions.keys()))
    distribution = distributions.pop(name)

    # Generate n draws from the distribution
    data = distribution.rvs(n)

    # Compute sample mean at each n
    sample_mean = np.empty(n)
    for i in range(n):
        sample_mean[i] = np.mean(data[:i+1])

    # Plot
    ax.plot(list(range(n)), data, 'o', color='grey', alpha=0.5)
    axlabel = '$\\bar X_n$ for $X_i \sim$' + name
```

```
        ax.plot(list(range(n)), sample_mean, 'g-', lw=3, alpha=0.6,
↪label=axlabel)
        m = distribution.mean()
        ax.plot(list(range(n)), [m] * n, 'k--', lw=1.5, label='$\mu$')
        ax.vlines(list(range(n)), m, data, lw=0.2)
        ax.legend(**legend_args, fontsize=12)

    plt.show()
```

$\bar{X}_n$ for $X_i \sim$ student's t with 10 degrees of freedom
μ
2
1
0
−1
−2
−3
−4
0
20
40
60
80
100

$\bar{X}_n$ for $X_i \sim$ exponential with $\lambda = 1$
μ
9
8
7
6
5
4
3
2
1
0
20
40
60
80
100

$\bar{X}_n$ for $X_i \sim \beta(2, 2)$
μ
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100

The three distributions are chosen at random from a selection stored in the dictionary `distributions`.

22.5 CLT

Next, we turn to the central limit theorem, which tells us about the distribution of the deviation between sample averages and population means.

22.5.1 Statement of the Theorem

The central limit theorem is one of the most remarkable results in all of mathematics.

In the classical IID setting, it tells us the following:

If the sequence $X_1, \dots, X_n$ is IID, with common mean μ and common variance $\sigma^2 \in (0, \infty)$, then

$$\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} N(0, \sigma^2) \quad \text{as} \quad n \to \infty \tag{5}$$

Here $\xrightarrow{d} N(0, \sigma^2)$ indicates convergence in distribution to a centered (i.e, zero mean) normal with standard deviation σ.

22.5.2 Intuition

The striking implication of the CLT is that for **any** distribution with finite second moment, the simple operation of adding independent copies **always** leads to a Gaussian curve.

A relatively simple proof of the central limit theorem can be obtained by working with characteristic functions (see, e.g., theorem 9.5.6 of [49]).

The proof is elegant but almost anticlimactic, and it provides surprisingly little intuition.

In fact, all of the proofs of the CLT that we know are similar in this respect.

Why does adding independent copies produce a bell-shaped distribution?

Part of the answer can be obtained by investigating the addition of independent Bernoulli random variables.

In particular, let X_i be binary, with $\mathbb{P}\{X_i = 0\} = \mathbb{P}\{X_i = 1\} = 0.5$, and let $X_1, \dots, X_n$ be independent.

Think of $X_i = 1$ as a "success", so that $Y_n = \sum_{i=1}^n X_i$ is the number of successes in n trials.

The next figure plots the probability mass function of Y_n for $n = 1, 2, 4, 8$

```
In [3]: fig, axes = plt.subplots(2, 2, figsize=(10, 6))
        plt.subplots_adjust(hspace=0.4)
        axes = axes.flatten()
        ns = [1, 2, 4, 8]
        dom = list(range(9))

        for ax, n in zip(axes, ns):
```

```
    b = binom(n, 0.5)
    ax.bar(dom, b.pmf(dom), alpha=0.6, align='center')
    ax.set(xlim=(-0.5, 8.5), ylim=(0, 0.55),
           xticks=list(range(9)), yticks=(0, 0.2, 0.4),
           title=f'$n = {n}$')

plt.show()
```

When $n = 1$, the distribution is flat — one success or no successes have the same probability.

When $n = 2$ we can either have 0, 1 or 2 successes.

Notice the peak in probability mass at the mid-point $k = 1$.

The reason is that there are more ways to get 1 success ("fail then succeed" or "succeed then fail") than to get zero or two successes.

Moreover, the two trials are independent, so the outcomes "fail then succeed" and "succeed then fail" are just as likely as the outcomes "fail then fail" and "succeed then succeed".

(If there was positive correlation, say, then "succeed then fail" would be less likely than "succeed then succeed")

Here, already we have the essence of the CLT: addition under independence leads probability mass to pile up in the middle and thin out at the tails.

For $n = 4$ and $n = 8$ we again get a peak at the "middle" value (halfway between the minimum and the maximum possible value).

The intuition is the same — there are simply more ways to get these middle outcomes.

If we continue, the bell-shaped curve becomes even more pronounced.

We are witnessing the binomial approximation of the normal distribution.

22.5.3 Simulation 1

Since the CLT seems almost magical, running simulations that verify its implications is one good way to build intuition.

To this end, we now perform the following simulation

1. Choose an arbitrary distribution F for the underlying observations X_i.

2. Generate independent draws of $Y_n := \sqrt{n}(\bar{X}_n - \mu)$.

3. Use these draws to compute some measure of their distribution — such as a histogram.

4. Compare the latter to $N(0, \sigma^2)$.

Here's some code that does exactly this for the exponential distribution $F(x) = 1 - e^{-\lambda x}$.

(Please experiment with other choices of F, but remember that, to conform with the conditions of the CLT, the distribution must have a finite second moment)

In [4]:
```
# Set parameters
n = 250                    # Choice of n
k = 100000                 # Number of draws of Y_n
distribution = expon(2)    # Exponential distribution, λ = 1/2
μ, s = distribution.mean(), distribution.std()

# Draw underlying RVs. Each row contains a draw of X_1,..,X_n
data = distribution.rvs((k, n))
# Compute mean of each row, producing k draws of \bar X_n
sample_means = data.mean(axis=1)
# Generate observations of Y_n
Y = np.sqrt(n) * (sample_means - μ)

# Plot
fig, ax = plt.subplots(figsize=(10, 6))
xmin, xmax = -3 * s, 3 * s
ax.set_xlim(xmin, xmax)
ax.hist(Y, bins=60, alpha=0.5, density=True)
xgrid = np.linspace(xmin, xmax, 200)
ax.plot(xgrid, norm.pdf(xgrid, scale=s), 'k-', lw=2, label='$N(0,
↪\sigma^2)$')
ax.legend()

plt.show()
```

Notice the absence of for loops — every operation is vectorized, meaning that the major calculations are all shifted to highly optimized C code.

The fit to the normal density is already tight and can be further improved by increasing `n`.

You can also experiment with other specifications of F.

22.5.4 Simulation 2

Our next simulation is somewhat like the first, except that we aim to track the distribution of $Y_n := \sqrt{n}(\bar{X}_n - \mu)$ as n increases.

In the simulation, we'll be working with random variables having $\mu = 0$.

Thus, when $n = 1$, we have $Y_1 = X_1$, so the first distribution is just the distribution of the underlying random variable.

For $n = 2$, the distribution of Y_2 is that of $(X_1 + X_2)/\sqrt{2}$, and so on.

What we expect is that, regardless of the distribution of the underlying random variable, the distribution of Y_n will smooth out into a bell-shaped curve.

The next figure shows this process for $X_i \sim f$, where f was specified as the convex combination of three different beta densities.

(Taking a convex combination is an easy way to produce an irregular shape for f)

In the figure, the closest density is that of Y_1, while the furthest is that of Y_5

```
In [5]: beta_dist = beta(2, 2)

        def gen_x_draws(k):
            """
            Returns a flat array containing k independent draws from the
            distribution of X, the underlying random variable.  This distribution
            is itself a convex combination of three beta distributions.
```

```
    """
    bdraws = beta_dist.rvs((3, k))
    # Transform rows, so each represents a different distribution
    bdraws[0, :] -= 0.5
    bdraws[1, :] += 0.6
    bdraws[2, :] -= 1.1
    # Set X[i] = bdraws[j, i], where j is a random draw from {0, 1, 2}
    js = np.random.randint(0, 2, size=k)
    X = bdraws[js, np.arange(k)]
    # Rescale, so that the random variable is zero mean
    m, sigma = X.mean(), X.std()
    return (X - m) / sigma

nmax = 5
reps = 100000
ns = list(range(1, nmax + 1))

# Form a matrix Z such that each column is reps independent draws of X
Z = np.empty((reps, nmax))
for i in range(nmax):
    Z[:, i] = gen_x_draws(reps)
# Take cumulative sum across columns
S = Z.cumsum(axis=1)
# Multiply j-th column by sqrt j
Y = (1 / np.sqrt(ns)) * S

# Plot
fig = plt.figure(figsize = (10, 6))
ax = fig.gca(projection='3d')

a, b = -3, 3
gs = 100
xs = np.linspace(a, b, gs)

# Build verts
greys = np.linspace(0.3, 0.7, nmax)
verts = []
for n in ns:
    density = gaussian_kde(Y[:, n-1])
    ys = density(xs)
    verts.append(list(zip(xs, ys)))

poly = PolyCollection(verts, facecolors=[str(g) for g in greys])
poly.set_alpha(0.85)
ax.add_collection3d(poly, zs=ns, zdir='x')

ax.set(xlim3d=(1, nmax), xticks=(ns), ylabel='$Y_n$', zlabel='$p(y_n)$',
       xlabel=("n"), yticks=((-3, 0, 3)), ylim3d=(a, b),
       zlim3d=(0, 0.4), zticks=((0.2, 0.4)))
ax.invert_xaxis()
# Rotates the plot 30 deg on z axis and 45 deg on x axis
ax.view_init(30, 45)
plt.show()
```

As expected, the distribution smooths out into a bell curve as n increases.

We leave you to investigate its contents if you wish to know more.

If you run the file from the ordinary IPython shell, the figure should pop up in a window that you can rotate with your mouse, giving different views on the density sequence.

22.5.5 The Multivariate Case

The law of large numbers and central limit theorem work just as nicely in multidimensional settings.

To state the results, let's recall some elementary facts about random vectors.

A random vector $\mathbf{X}$ is just a sequence of k random variables $(X_1, \dots, X_k)$.

Each realization of $\mathbf{X}$ is an element of $\mathbb{R}^k$.

A collection of random vectors $\mathbf{X}_1, \dots, \mathbf{X}_n$ is called independent if, given any n vectors $\mathbf{x}_1, \dots, \mathbf{x}_n$ in $\mathbb{R}^k$, we have

$$\mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1, \dots, \mathbf{X}_n \leq \mathbf{x}_n\} = \mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1\} \times \cdots \times \mathbb{P}\{\mathbf{X}_n \leq \mathbf{x}_n\}$$

(The vector inequality $\mathbf{X} \leq \mathbf{x}$ means that $X_j \leq x_j$ for $j = 1, \dots, k$)

Let $\mu_j := \mathbb{E}[X_j]$ for all $j = 1, \dots, k$.

The expectation $\mathbb{E}[\mathbf{X}]$ of $\mathbf{X}$ is defined to be the vector of expectations:

$$\mathbb{E}[\mathbf{X}] := \begin{pmatrix} \mathbb{E}[X_1] \\ \mathbb{E}[X_2] \\ \vdots \\ \mathbb{E}[X_k] \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_k \end{pmatrix} =: \mu$$

The *variance-covariance matrix* of random vector $\mathbf{X}$ is defined as

$$\mathrm{Var}[\mathbf{X}] := \mathbb{E}[(\mathbf{X} - \mu)(\mathbf{X} - \mu)']$$

Expanding this out, we get

$$\mathrm{Var}[\mathbf{X}] = \begin{pmatrix} \mathbb{E}[(X_1 - \mu_1)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_1 - \mu_1)(X_k - \mu_k)] \\ \mathbb{E}[(X_2 - \mu_2)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_2 - \mu_2)(X_k - \mu_k)] \\ \vdots & \vdots & \vdots \\ \mathbb{E}[(X_k - \mu_k)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_k - \mu_k)(X_k - \mu_k)] \end{pmatrix}$$

The j, k-th term is the scalar covariance between X_j and X_k.

With this notation, we can proceed to the multivariate LLN and CLT.

Let $\mathbf{X}_1, \dots, \mathbf{X}_n$ be a sequence of independent and identically distributed random vectors, each one taking values in $\mathbb{R}^k$.

Let μ be the vector $\mathbb{E}[\mathbf{X}_i]$, and let Σ be the variance-covariance matrix of $\mathbf{X}_i$.

Interpreting vector addition and scalar multiplication in the usual way (i.e., pointwise), let

$$\bar{\mathbf{X}}_n := \frac{1}{n} \sum_{i=1}^{n} \mathbf{X}_i$$

In this setting, the LLN tells us that

$$\mathbb{P}\left\{\bar{\mathbf{X}}_n \to \mu \text{ as } n \to \infty\right\} = 1 \tag{6}$$

Here $\bar{\mathbf{X}}_n \to \mu$ means that $\|\bar{\mathbf{X}}_n - \mu\| \to 0$, where $\|\cdot\|$ is the standard Euclidean norm.

The CLT tells us that, provided Σ is finite,

$$\sqrt{n}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \Sigma) \quad \text{as} \quad n \to \infty \tag{7}$$

22.6 Exercises

22.6.1 Exercise 1

One very useful consequence of the central limit theorem is as follows.

Assume the conditions of the CLT as stated above.

If $g \colon \mathbb{R} \to \mathbb{R}$ is differentiable at μ and $g'(\mu) \neq 0$, then

$$\sqrt{n}\{g(\bar{X}_n) - g(\mu)\} \xrightarrow{d} N(0, g'(\mu)^2 \sigma^2) \quad \text{as} \quad n \to \infty \tag{8}$$

This theorem is used frequently in statistics to obtain the asymptotic distribution of estimators — many of which can be expressed as functions of sample means.

(These kinds of results are often said to use the "delta method")

The proof is based on a Taylor expansion of g around the point μ.

Taking the result as given, let the distribution F of each X_i be uniform on $[0, \pi/2]$ and let $g(x) = \sin(x)$.

Derive the asymptotic distribution of $\sqrt{n}\{g(\bar{X}_n) - g(\mu)\}$ and illustrate convergence in the same spirit as the program `illustrate_clt.py` discussed above.

What happens when you replace $[0, \pi/2]$ with $[0, \pi]$?

What is the source of the problem?

22.6.2 Exercise 2

Here's a result that's often used in developing statistical tests, and is connected to the multivariate central limit theorem.

If you study econometric theory, you will see this result used again and again.

Assume the setting of the multivariate CLT discussed above, so that

1. $\mathbf{X}_1, \dots, \mathbf{X}_n$ is a sequence of IID random vectors, each taking values in $\mathbb{R}^k$.
2. $\mu := \mathbb{E}[\mathbf{X}_i]$, and Σ is the variance-covariance matrix of $\mathbf{X}_i$.
3. The convergence

$$\sqrt{n}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \Sigma) \tag{9}$$

is valid.

In a statistical setting, one often wants the right-hand side to be **standard** normal so that confidence intervals are easily computed.

This normalization can be achieved on the basis of three observations.

First, if $\mathbf{X}$ is a random vector in $\mathbb{R}^k$ and $\mathbf{A}$ is constant and $k \times k$, then

$$\operatorname{Var}[\mathbf{AX}] = \mathbf{A} \operatorname{Var}[\mathbf{X}] \mathbf{A}'$$

Second, by the continuous mapping theorem, if $\mathbf{Z}_n \xrightarrow{d} \mathbf{Z}$ in $\mathbb{R}^k$ and $\mathbf{A}$ is constant and $k \times k$, then

$$\mathbf{AZ}_n \xrightarrow{d} \mathbf{AZ}$$

Third, if $\mathbf{S}$ is a $k \times k$ symmetric positive definite matrix, then there exists a symmetric positive definite matrix $\mathbf{Q}$, called the inverse square root of $\mathbf{S}$, such that

$$\mathbf{QSQ}' = \mathbf{I}$$

Here $\mathbf{I}$ is the $k \times k$ identity matrix.

Putting these things together, your first exercise is to show that if $\mathbf{Q}$ is the inverse square root of , then

$$\mathbf{Z}_n := \sqrt{n}\mathbf{Q}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} \mathbf{Z} \sim N(\mathbf{0}, \mathbf{I})$$

Applying the continuous mapping theorem one more time tells us that

$$\|\mathbf{Z}_n\|^2 \xrightarrow{d} \|\mathbf{Z}\|^2$$

Given the distribution of $\mathbf{Z}$, we conclude that

$$n\|\mathbf{Q}(\bar{\mathbf{X}}_n - \mu)\|^2 \xrightarrow{d} \chi^2(k) \tag{10}$$

where $\chi^2(k)$ is the chi-squared distribution with k degrees of freedom.

(Recall that k is the dimension of $\mathbf{X}_i$, the underlying random vectors)

Your second exercise is to illustrate the convergence in (10) with a simulation.

In doing so, let

$$\mathbf{X}_i := \begin{pmatrix} W_i \\ U_i + W_i \end{pmatrix}$$

where

- each W_i is an IID draw from the uniform distribution on $[-1, 1]$.
- each U_i is an IID draw from the uniform distribution on $[-2, 2]$.
- U_i and W_i are independent of each other.

Hints:

1. `scipy.linalg.sqrtm(A)` computes the square root of `A`. You still need to invert it.
2. You should be able to work out Σ from the preceding information.

22.7 Solutions

22.7.1 Exercise 1

Here is one solution

In [6]:
```
"""
Illustrates the delta method, a consequence of the central limit theorem.
"""

# Set parameters
n = 250
replications = 100000
distribution = uniform(loc=0, scale=(np.pi / 2))
μ, s = distribution.mean(), distribution.std()

g = np.sin
g_prime = np.cos
```

```
# Generate obs of sqrt{n} (g(X_n) - g(μ))
data = distribution.rvs((replications, n))
sample_means = data.mean(axis=1)  # Compute mean of each row
error_obs = np.sqrt(n) * (g(sample_means) - g(μ))

# Plot
asymptotic_sd = g_prime(μ) * s
fig, ax = plt.subplots(figsize=(10, 6))
xmin = -3 * g_prime(μ) * s
xmax = -xmin
ax.set_xlim(xmin, xmax)
ax.hist(error_obs, bins=60, alpha=0.5, density=True)
xgrid = np.linspace(xmin, xmax, 200)
lb = "$N(0, g'(\mu)^2  \sigma^2)$"
ax.plot(xgrid, norm.pdf(xgrid, scale=asymptotic_sd), 'k-', lw=2, label=lb)
ax.legend()
plt.show()
```

What happens when you replace $[0, \pi/2]$ with $[0, \pi]$?

In this case, the mean μ of this distribution is $\pi/2$, and since $g' = \cos$, we have $g'(\mu) = 0$.

Hence the conditions of the delta theorem are not satisfied.

22.7.2 Exercise 2

First we want to verify the claim that

$$\sqrt{n}\mathbf{Q}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \mathbf{I})$$

This is straightforward given the facts presented in the exercise.

Let

$$\mathbf{Y}_n := \sqrt{n}(\bar{\mathbf{X}}_n - \mu) \quad \text{and} \quad \mathbf{Y} \sim N(\mathbf{0}, \Sigma)$$

By the multivariate CLT and the continuous mapping theorem, we have

$$\mathbf{Q}\mathbf{Y}_n \xrightarrow{d} \mathbf{Q}\mathbf{Y}$$

Since linear combinations of normal random variables are normal, the vector $\mathbf{Q}\mathbf{Y}$ is also normal.

Its mean is clearly $\mathbf{0}$, and its variance-covariance matrix is

$$\text{Var}[\mathbf{Q}\mathbf{Y}] = \mathbf{Q}\text{Var}[\mathbf{Y}]\mathbf{Q}' = \mathbf{Q}\Sigma\mathbf{Q}' = \mathbf{I}$$

In conclusion, $\mathbf{Q}\mathbf{Y}_n \xrightarrow{d} \mathbf{Q}\mathbf{Y} \sim N(\mathbf{0}, \mathbf{I})$, which is what we aimed to show.

Now we turn to the simulation exercise.

Our solution is as follows

In [7]:
```
# Set parameters
n = 250
replications = 50000
dw = uniform(loc=-1, scale=2)  # Uniform(-1, 1)
du = uniform(loc=-2, scale=4)  # Uniform(-2, 2)
sw, su = dw.std(), du.std()
vw, vu = sw**2, su**2
Σ = ((vw, vw), (vw, vw + vu))
Σ = np.array(Σ)

# Compute Σ^{-1/2}
Q = inv(sqrtm(Σ))

# Generate observations of the normalized sample mean
error_obs = np.empty((2, replications))
for i in range(replications):
    # Generate one sequence of bivariate shocks
    X = np.empty((2, n))
    W = dw.rvs(n)
    U = du.rvs(n)
    # Construct the n observations of the random vector
    X[0, :] = W
    X[1, :] = W + U
    # Construct the i-th observation of Y_n
    error_obs[:, i] = np.sqrt(n) * X.mean(axis=1)

# Premultiply by Q and then take the squared norm
temp = Q @ error_obs
chisq_obs = np.sum(temp**2, axis=0)

# Plot
fig, ax = plt.subplots(figsize=(10, 6))
xmax = 8
ax.set_xlim(0, xmax)
xgrid = np.linspace(0, xmax, 200)
lb = "Chi-squared with 2 degrees of freedom"
```

```
ax.plot(xgrid, chi2.pdf(xgrid, 2), 'k-', lw=2, label=lb)
ax.legend()
ax.hist(chisq_obs, bins=50, density=True)
plt.show()
```

Chapter 23

Heavy-Tailed Distributions

23.1 Contents

- Overview 23.2
- Visual Comparisons 23.3
- Failure of the LLN 23.4
- Classifying Tail Properties 23.5
- Exercises 23.6
- Solutions 23.7

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

```
In [1]: !pip install --upgrade quantecon
        !pip install --upgrade yfinance
```

23.2 Overview

Most commonly used probability distributions in classical statistics and the natural sciences have either bounded support or light tails.

When a distribution is light-tailed, extreme observations are rare and draws tend not to deviate too much from the mean.

Having internalized these kinds of distributions, many researchers and practitioners use rules of thumb such as "outcomes more than four or five standard deviations from the mean can safely be ignored."

However, some distributions encountered in economics have far more probability mass in the tails than distributions like the normal distribution.

With such **heavy-tailed** distributions, what would be regarded as extreme outcomes for someone accustomed to thin tailed distributions occur relatively frequently.

Examples of heavy-tailed distributions observed in economic and financial settings include

- the income distributions and the wealth distribution (see, e.g., [159], [17]),
- the firm size distribution ([13], [59]}),
- the distribution of returns on holding assets over short time horizons ([112], [128]), and
- the distribution of city sizes ([135], [59]).

These heavy tails turn out to be important for our understanding of economic outcomes.

As one example, the heaviness of the tail in the wealth distribution is one natural measure of inequality.

It matters for taxation and redistribution policies, as well as for flow-on effects for productivity growth, business cycles, and political economy

- see, e.g., [4], [63], [24] or [5].

This lecture formalizes some of the concepts introduced above and reviews the key ideas.

Let's start with some imports:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

The following two lines can be added to avoid an annoying FutureWarning, and prevent a specific compatibility issue between pandas and matplotlib from causing problems down the line:

```
In [3]: from pandas.plotting import register_matplotlib_converters
        register_matplotlib_converters()
```

23.3 Visual Comparisons

One way to build intuition on the difference between light and heavy tails is to plot independent draws and compare them side-by-side.

23.3.1 A Simulation

The figure below shows a simulation. (You will be asked to replicate it in the exercises.)

The top two subfigures each show 120 independent draws from the normal distribution, which is light-tailed.

The bottom subfigure shows 120 independent draws from the Cauchy distribution, which is

heavy-tailed.

In the top subfigure, the standard deviation of the normal distribution is 2, and the draws are clustered around the mean.

In the middle subfigure, the standard deviation is increased to 12 and, as expected, the amount of dispersion rises.

The bottom subfigure, with the Cauchy draws, shows a different pattern: tight clustering around the mean for the great majority of observations, combined with a few sudden large deviations from the mean.

This is typical of a heavy-tailed distribution.

23.3.2 Heavy Tails in Asset Returns

Next let's look at some financial data.

Our aim is to plot the daily change in the price of Amazon (AMZN) stock for the period from 1st January 2015 to 1st November 2019.

This equates to daily returns if we set dividends aside.

The code below produces the desired plot using Yahoo financial data via the `yfinance` library.

In [4]:
```
import yfinance as yf
import pandas as pd

s = yf.download('AMZN', '2015-1-1', '2019-11-1')['Adj Close']

r = s.pct_change()

fig, ax = plt.subplots()

ax.plot(r, linestyle='', marker='o', alpha=0.5, ms=4)
ax.vlines(r.index, 0, r.values, lw=0.2)

ax.set_ylabel('returns', fontsize=12)
ax.set_xlabel('date', fontsize=12)

plt.show()
```

```
[*********************100%***********************]  1 of 1 completed
```

Five of the 1217 observations are more than 5 standard deviations from the mean.

Overall, the figure is suggestive of heavy tails, although not to the same degree as the Cauchy distribution the figure above.

If, however, one takes tick-by-tick data rather daily data, the heavy-tailedness of the distribution increases further.

23.4 Failure of the LLN

One impact of heavy tails is that sample averages can be poor estimators of the underlying mean of the distribution.

To understand this point better, recall our earlier discussion of the Law of Large Numbers, which considered IID $X_1, \dots, X_n$ with common distribution F

If $\mathbb{E}|X_i|$ is finite, then the sample mean $\bar{X}_n := \frac{1}{n}\sum_{i=1}^n X_i$ satisfies

$$\mathbb{P}\left\{\bar{X}_n \to \mu \text{ as } n \to \infty\right\} = 1 \tag{1}$$

where $\mu := \mathbb{E}X_i = \int xF(x)$ is the common mean of the sample.

The condition $\mathbb{E}|X_i| = \int |x|F(x) < \infty$ holds in most cases but can fail if the distribution F is very heavy tailed.

For example, it fails for the Cauchy distribution

Let's have a look at the behavior of the sample mean in this case, and see whether or not the LLN is still valid.

```
In [5]: from scipy.stats import cauchy
```

```
np.random.seed(1234)
N = 1_000

distribution = cauchy()

fig, ax = plt.subplots()
data = distribution.rvs(N)

# Compute sample mean at each n
sample_mean = np.empty(N)
for n in range(1, N):
    sample_mean[n] = np.mean(data[:n])

# Plot
ax.plot(range(N), sample_mean, alpha=0.6, label='$\\bar X_n$')

ax.plot(range(N), np.zeros(N), 'k--', lw=0.5)
ax.legend()

plt.show()
```

The sequence shows no sign of converging.

Will convergence occur if we take n even larger?

The answer is no.

To see this, recall that the characteristic function of the Cauchy distribution is

$$\phi(t) = \mathbb{E}e^{itX} = \int e^{itx} f(x)dx = e^{-|t|} \tag{2}$$

Using independence, the characteristic function of the sample mean becomes

$$\begin{aligned}\mathbb{E}e^{it\bar{X}_n} &= \mathbb{E}\exp\left\{i\frac{t}{n}\sum_{j=1}^{n}X_j\right\}\\ &= \mathbb{E}\prod_{j=1}^{n}\exp\left\{i\frac{t}{n}X_j\right\}\\ &= \prod_{j=1}^{n}\mathbb{E}\exp\left\{i\frac{t}{n}X_j\right\} = [\phi(t/n)]^n\end{aligned}$$

In view of (2), this is just $e^{-|t|}$.

Thus, in the case of the Cauchy distribution, the sample mean itself has the very same Cauchy distribution, regardless of n!

In particular, the sequence $\bar{X}_n$ does not converge to any point.

23.5 Classifying Tail Properties

To keep our discussion precise, we need some definitions concerning tail properties.

We will focus our attention on the right hand tails of nonnegative random variables and their distributions.

The definitions for left hand tails are very similar and we omit them to simplify the exposition.

23.5.1 Light and Heavy Tails

A distribution F on $\mathbb{R}_+$ is called **heavy-tailed** if

$$\int_0^\infty \exp(tx)F(dx) = \infty \quad \text{for all } t > 0. \tag{3}$$

We say that a nonnegative random variable X is **heavy-tailed** if its distribution $F(x) := \mathbb{P}\{X \leq x\}$ is heavy-tailed.

This is equivalent to stating that its **moment generating function** $m(t) := \mathbb{E}\exp(tX)$ is infinite for all $t > 0$.

- For example, the lognormal distribution is heavy-tailed because its moment generating function is infinite everywhere on $(0, \infty)$.

A distribution F on $\mathbb{R}_+$ is called **light-tailed** if it is not heavy-tailed.

A nonnegative random variable X is **light-tailed** if its distribution F is light-tailed.

- Example: Every random variable with bounded support is light-tailed. (Why?)
- Example: If X has the exponential distribution, with cdf $F(x) = 1 - \exp(-\lambda x)$ for some $\lambda > 0$, then its moment generating function is finite whenever $t < \lambda$. Hence X is light-tailed.

One can show that if X is light-tailed, then all of its moments are finite.

The contrapositive is that if some moment is infinite, then X is heavy-tailed.

The latter condition is not necessary, however.

- Example: the lognormal distribution is heavy-tailed but every moment is finite.

23.5.2 Pareto Tails

One specific class of heavy-tailed distributions has been found repeatedly in economic and social phenomena: the class of so-called power laws.

Specifically, given $\alpha > 0$, a nonnegative random variable X is said to have a **Pareto tail** with **tail index** α if

$$\lim_{x \to \infty} x^\alpha \, \mathbb{P}\{X > x\} = c. \tag{4}$$

Evidently (4) implies the existence of positive constants b and $\bar{x}$ such that $\mathbb{P}\{X > x\} \geq bx^{-\alpha}$ whenever $x \geq \bar{x}$.

The implication is that $\mathbb{P}\{X > x\}$ converges to zero no faster than $x^{-\alpha}$.

In some sources, a random variable obeying (4) is said to have a **power law tail**.

The primary example is the **Pareto distribution**, which has distribution

$$F(x) = \begin{cases} 1 - (\bar{x}/x)^\alpha & \text{if } x \geq \bar{x} \\ 0 & \text{if } x < \bar{x} \end{cases} \tag{5}$$

for some positive constants $\bar{x}$ and α.

It is easy to see that if $X \sim F$, then $\mathbb{P}\{X > x\}$ satisfies (4).

Thus, in line with the terminology, Pareto distributed random variables have a Pareto tail.

23.5.3 Rank-Size Plots

One graphical technique for investigating Pareto tails and power laws is the so-called **rank-size plot**.

This kind of figure plots log size against log rank of the population (i.e., location in the population when sorted from smallest to largest).

Often just the largest 5 or 10% of observations are plotted.

For a sufficiently large number of draws from a Pareto distribution, the plot generates a straight line. For distributions with thinner tails, the data points are concave.

A discussion of why this occurs can be found in [122].

The figure below provides one example, using simulated data.

The rank-size plots shows draws from three different distributions: folded normal, chi-squared with 1 degree of freedom and Pareto.

The Pareto sample produces a straight line, while the lines produced by the other samples are concave.

You are asked to reproduce this figure in the exercises.

23.6 Exercises

23.6.1 Exercise 1

Replicate the figure presented above that compares normal and Cauchy draws.

Use `np.random.seed(11)` to set the seed.

23.6.2 Exercise 2

Prove: If X has a Pareto tail with tail index α, then $\mathbb{E}[X^r] = \infty$ for all $r \geq \alpha$.

23.6.3 Exercise 3

Repeat exercise 1, but replace the three distributions (two normal, one Cauchy) with three Pareto distributions using different choices of α.

For α, try 1.15, 1.5 and 1.75.

Use `np.random.seed(11)` to set the seed.

23.6.4 Exercise 4

Replicate the rank-size plot figure presented above.

For each sample that you plot, you will need to reverse sort it, from largest to smallest, in order to generate the plot.

Use `np.random.seed(13)` to set the seed.

23.6.5 Exercise 5

There is an ongoing argument about whether the firm size distribution should be modeled as a Pareto distribution or a lognormal distribution (see, e.g., [58], [98] or [146]).

This sounds esoteric but has real implications for a variety of economic phenomena.

To illustrate this fact in a simple way, let us consider an economy with 100,000 firms, an interest rate of `r = 0.05` and a corporate tax rate of 15%.

Your task is to estimate the present discounted value of projected corporate tax revenue over the next 10 years.

Because we are forecasting, we need a model.

We will suppose that

1. the number of firms and the firm size distribution (measured in profits) remain fixed and
2. the firm size distribution is either lognormal or Pareto.

Present discounted value of tax revenue will be estimated by

1. generating 100,000 draws of firm profit from the firm size distribution,
2. multiplying by the tax rate, and
3. summing the results with discounting to obtain present value.

The Pareto distribution is assumed to take the form (5) with $\bar{x} = 1$ and $\alpha = 1.05$.

(The value the tail index α is plausible given the data [59].)

To make the lognormal option as similar as possible to the Pareto option, choose its parameters such that the mean and median of both distributions are the same.

Note that, for each distribution, your estimate of tax revenue will be random because it is based on a finite number of draws.

To take this into account, generate 100 replications (evaluations of tax revenue) for each of the two distributions and compare the two samples by

- producing a violin plot visualizing the two samples side-by-side and
- printing the mean and standard deviation of both samples.

For the seed use `np.random.seed(1234)`.

What differences do you observe?

(Note: a better approach to this problem would be to model firm dynamics and try to track individual firms given the current distribution. We will discuss firm dynamics in later lectures.)

23.7 Solutions

23.7.1 Exercise 1

```
In [6]: n = 120
        np.random.seed(11)

        fig, axes = plt.subplots(3, 1, figsize=(6, 12))

        for ax in axes:
            ax.set_ylim((-120, 120))

        s_vals = 2, 12

        for ax, s in zip(axes[:2], s_vals):
            data = np.random.randn(n) * s
           ax.plot(list(range(n)), data, linestyle='', marker='o', alpha=0.5, ms=4)
            ax.vlines(list(range(n)), 0, data, lw=0.2)
            ax.set_title(f"draws from $N(0, \sigma^2)$ with $\sigma = {s}$",
↪fontsize=11)

        ax = axes[2]
        distribution = cauchy()
        data = distribution.rvs(n)
        ax.plot(list(range(n)), data, linestyle='', marker='o', alpha=0.5, ms=4)
        ax.vlines(list(range(n)), 0, data, lw=0.2)
        ax.set_title(f"draws from the Cauchy distribution", fontsize=11)

        plt.subplots_adjust(hspace=0.25)

        plt.show()
```

draws from $N(0, \sigma^2)$ with $\sigma = 2$
100
50
0
−50
−100
0
20
40
60
80
100
120

draws from $N(0, \sigma^2)$ with $\sigma = 12$
100
50
0
−50
−100
0
20
40
60
80
100
120

draws from the Cauchy distribution
100
50
0
−50
−100
0
20
40
60
80
100
120

23.7.2 Exercise 2

Let X have a Pareto tail with tail index α and let F be its cdf.

Fix $r \geq \alpha$.

As discussed after (4), we can take positive constants b and $\bar{x}$ such that

$$\mathbb{P}\{X > x\} \geq bx^{-\alpha} \text{ whenever } x \geq \bar{x}$$

But then

$$\mathbb{E}X^r = r \int_0^\infty x^{r-1}\mathbb{P}\{X > x\}x \geq r \int_0^{\bar{x}} x^{r-1}\mathbb{P}\{X > x\}x + r \int_{\bar{x}}^\infty x^{r-1}bx^{-\alpha}x.$$

We know that $\int_{\bar{x}}^\infty x^{r-\alpha-1}x = \infty$ whenever $r - \alpha - 1 \geq -1$.

Since $r \geq \alpha$, we have $\mathbb{E}X^r = \infty$.

23.7.3 Exercise 3

In [7]:
```
from scipy.stats import pareto

np.random.seed(11)

n = 120
alphas = [1.15, 1.50, 1.75]

fig, axes = plt.subplots(3, 1, figsize=(6, 8))

for (a, ax) in zip(alphas, axes):
    ax.set_ylim((-5, 50))
    data = pareto.rvs(size=n, scale=1, b=a)
   ax.plot(list(range(n)), data, linestyle='', marker='o', alpha=0.5, ms=4)
    ax.vlines(list(range(n)), 0, data, lw=0.2)
    ax.set_title(f"Pareto draws with $\\alpha = {a}$", fontsize=11)

plt.subplots_adjust(hspace=0.4)

plt.show()
```

23.7.4 Exercise 4

First we will create a function and then generate the plot

```
In [8]: def rank_size_data(data, c=1.0):
            """
            Generate rank-size data corresponding to distribution data.

                * data is array like
                * c is a float indicating the top (c x 100)% of the
                  distribution
            """
            w = - np.sort(- data)                    # Reverse sort
```

```
        w = w[:int(len(w) * c)]                # extract top c%
        rank_data = np.log(np.arange(len(w)) + 1)
        size_data = np.log(w)
        return rank_data, size_data

    fig, axes = plt.subplots(3, 1, figsize=(6, 8))
    axes = axes.flatten()

    sample_size = 1000
    np.random.seed(13)
    z = np.random.randn(sample_size)

    data_1 = np.abs(z)
    data_2 = np.exp(z)
    data_3 = np.exp(np.random.exponential(scale=1.0, size=sample_size))

    data_list = [data_1, data_2, data_3]
    labels = ['$|z|$', '$\exp(z)$', 'Pareto with tail index $1.0$']


    for data, label, ax in zip(data_list, labels, axes):
        rank_data, size_data = rank_size_data(data)
        ax.plot(rank_data, size_data, 'o', markersize=3.0, alpha=0.5,
↪label=label)

        ax.set_xlabel("log rank")
        ax.set_ylabel("log size")
        ax.legend()



    fig.subplots_adjust(hspace=0.4)

    plt.show()
```

23.7.5 Exercise 5

To do the exercise, we need to choose the parameters μ and σ of the lognormal distribution to match the mean and median of the Pareto distribution.

Here we understand the lognormal distribution as that of the random variable $\exp(\mu + \sigma Z)$ when Z is standard normal.

The mean and median of the Pareto distribution (5) with $\bar{x} = 1$ are

$$\text{mean} \ = \frac{\alpha}{\alpha - 1} \quad \text{and} \quad \text{median} \ = 2^{1/\alpha}$$

Using the corresponding expressions for the lognormal distribution leads us to the equations

$$\frac{\alpha}{\alpha - 1} = \exp(\mu + \sigma^2/2) \quad \text{and} \quad 2^{1/\alpha} = \exp(\mu)$$

which we solve for μ and σ given $\alpha = 1.05$

Here is code that generates the two samples, produces the violin plot and prints the mean and standard deviation of the two samples.

```
In [9]: num_firms = 100_000
        num_years = 10
        tax_rate = 0.15
        r = 0.05

        β = 1 / (1 + r)    # discount factor

        x_bar = 1.0
        α = 1.05

        def pareto_rvs(n):
            "Uses a standard method to generate Pareto draws."
            u = np.random.uniform(size=n)
            y = x_bar / (u**(1/α))
            return y
```

Let's compute the lognormal parameters:

```
In [10]: μ = np.log(2) / α
         σ_sq = 2 * (np.log(α/(α - 1)) - np.log(2)/α)
         σ = np.sqrt(σ_sq)
```

Here's a function to compute a single estimate of tax revenue for a particular choice of distribution `dist`.

```
In [11]: def tax_rev(dist):
             tax_raised = 0
             for t in range(num_years):
                 if dist == 'pareto':
                     π = pareto_rvs(num_firms)
                 else:
                     π = np.exp(μ + σ * np.random.randn(num_firms))
                 tax_raised += β**t * np.sum(π * tax_rate)
             return tax_raised
```

Now let's generate the violin plot.

```
In [12]: num_reps = 100
         np.random.seed(1234)

         tax_rev_lognorm = np.empty(num_reps)
         tax_rev_pareto = np.empty(num_reps)

         for i in range(num_reps):
             tax_rev_pareto[i] = tax_rev('pareto')
             tax_rev_lognorm[i] = tax_rev('lognorm')
```

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

data = tax_rev_pareto, tax_rev_lognorm

ax.violinplot(data)

plt.show()
```

Finally, let's print the means and standard deviations.

```
In [13]: tax_rev_pareto.mean(), tax_rev_pareto.std()

Out[13]: (1458729.0546623734, 406089.3613661567)

In [14]: tax_rev_lognorm.mean(), tax_rev_lognorm.std()

Out[14]: (2556174.8615230713, 25586.44456513963)
```

Looking at the output of the code, our main conclusion is that the Pareto assumption leads to a lower mean and greater dispersion.

Part V

Introduction to Dynamics

Chapter 24

Dynamics in One Dimension

24.1 Contents

- Overview 24.2
- Some Definitions 24.3
- Graphical Analysis 24.4
- Exercises 24.5
- Solutions 24.6

24.2 Overview

In this lecture we give a quick introduction to discrete time dynamics in one dimension.

In one-dimensional models, the state of the system is described by a single variable.

Although most interesting dynamic models have two or more state variables, the one-dimensional setting is a good place to learn the foundations of dynamics and build intuition.

Let's start with some standard imports:

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

24.3 Some Definitions

This section sets out the objects of interest and the kinds of properties we study.

24.3.1 Difference Equations

A **time homogeneous first order difference equation** is an equation of the form

$$x_{t+1} = g(x_t) \tag{1}$$

where g is a function from some subset S of $\mathbb{R}$ to itself.

Here S is called the **state space** and x is called the **state variable**.

In the definition,

- time homogeneity means that g is the same at each time t
- first order means dependence on only one lag (i.e., earlier states such as x_{t-1} do not enter into (1)).

If $x_0 \in S$ is given, then (1) recursively defines the sequence

$$x_0, \quad x_1 = g(x_0), \quad x_2 = g(x_1) = g(g(x_0)), \quad \text{etc.} \tag{2}$$

This sequence is called the **trajectory** of x_0 under g.

If we define g^n to be n compositions of g with itself, then we can write the trajectory more simply as $x_t = g^t(x_0)$ for $t \geq 0$.

24.3.2 Example: A Linear Model

One simple example is the **linear difference equation**

$$x_{t+1} = ax_t + b, \qquad S = \mathbb{R}$$

where a, b are fixed constants.

In this case, given x_0, the trajectory (2) is

$$x_0, \quad ax_0 + b, \quad a^2 x_0 + ab + b, \quad \text{etc.} \tag{3}$$

Continuing in this way, and using our knowledge of geometric series, we find that, for any $t \geq 0$,

$$x_t = a^t x_0 + b\frac{1 - a^t}{1 - a} \tag{4}$$

This is about all we need to know about the linear model.

We have an exact expression for x_t for all t and hence a full understanding of the dynamics.

Notice in particular that $|a| < 1$, then, by (4), we have

$$x_t \to \frac{b}{1 - a} \text{ as } t \to \infty \tag{5}$$

regardless of x_0

This is an example of what is called global stability, a topic we return to below.

24.3.3 Example: A Nonlinear Model

In the linear example above, we obtained an exact analytical expression for x_t in terms of arbitrary t and x_0.

This made analysis of dynamics very easy.

When models are nonlinear, however, the situation can be quite different.

For example, recall how we previously studied the law of motion for the Solow growth model, a simplified version of which is

$$k_{t+1} = szk_t^\alpha + (1-\delta)k_t \tag{6}$$

Here k is capital stock and s, z, α, δ are positive parameters with $0 < \alpha, \delta < 1$.

If you try to iterate like we did in (3), you will find that the algebra gets messy quickly.

Analyzing the dynamics of this model requires a different method (see below).

24.3.4 Stability

A **steady state** of the difference equation $x_{t+1} = g(x_t)$ is a point x^* in S such that $x^* = g(x^*)$.

In other words, x^* is a **fixed point** of the function g in S.

For example, for the linear model $x_{t+1} = ax_t + b$, you can use the definition to check that

- $x^* := b/(1-a)$ is a steady state whenever $a \neq 1$.
- if $a = 1$ and $b = 0$, then every $x \in \mathbb{R}$ is a steady state.
- if $a = 1$ and $b \neq 0$, then the linear model has no steady state in $\mathbb{R}$.

A steady state x^* of $x_{t+1} = g(x_t)$ is called **globally stable** if, for all $x_0 \in S$,

$$x_t = g^t(x_0) \to x^* \text{ as } t \to \infty$$

For example, in the linear model $x_{t+1} = ax_t + b$ with $a \neq 1$, the steady state x^*

- is globally stable if $|a| < 1$ and
- fails to be globally stable otherwise.

This follows directly from (4).

A steady state x^* of $x_{t+1} = g(x_t)$ is called **locally stable** if there exists an $\epsilon > 0$ such that

$$|x_0 - x^*| < \epsilon \implies x_t = g^t(x_0) \to x^* \text{ as } t \to \infty$$

Obviously every globally stable steady state is also locally stable.

We will see examples below where the converse is not true.

24.4 Graphical Analysis

As we saw above, analyzing the dynamics for nonlinear models is nontrivial.

There is no single way to tackle all nonlinear models.

However, there is one technique for one-dimensional models that provides a great deal of intuition.

This is a graphical approach based on **45 degree diagrams**.

Let's look at an example: the Solow model with dynamics given in (6).

We begin with some plotting code that you can ignore at first reading.

The function of the code is to produce 45 degree diagrams and time series plots.

In [2]:
```
def subplots(fs):
    "Custom subplots with axes throught the origin"
    fig, ax = plt.subplots(figsize=fs)

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

    return fig, ax


def plot45(g, xmin, xmax, x0, num_arrows=6, var='x'):

    xgrid = np.linspace(xmin, xmax, 200)

    fig, ax = subplots((6.5, 6))
    ax.set_xlim(xmin, xmax)
    ax.set_ylim(xmin, xmax)

    hw = (xmax - xmin) * 0.01
    hl = 2 * hw
    arrow_args = dict(fc="k", ec="k", head_width=hw,
            length_includes_head=True, lw=1,
            alpha=0.6, head_length=hl)

    ax.plot(xgrid, g(xgrid), 'b-', lw=2, alpha=0.6, label='g')
    ax.plot(xgrid, xgrid, 'k-', lw=1, alpha=0.7, label='45')

    x = x0
    xticks = [xmin]
    xtick_labels = [xmin]

    for i in range(num_arrows):
        if i == 0:
            ax.arrow(x, 0.0, 0.0, g(x), **arrow_args) # x, y, dx, dy
        else:
            ax.arrow(x, x, 0.0, g(x) - x, **arrow_args)
            ax.plot((x, x), (0, x), 'k', ls='dotted')

        ax.arrow(x, g(x), g(x) - x, 0, **arrow_args)
        xticks.append(x)
        xtick_labels.append(r'${}_{}$'.format(var, str(i)))

        x = g(x)
        xticks.append(x)
        xtick_labels.append(r'${}_{}$'.format(var, str(i+1)))
        ax.plot((x, x), (0, x), 'k-', ls='dotted')

    xticks.append(xmax)
    xtick_labels.append(xmax)
```

```
    ax.set_xticks(xticks)
    ax.set_yticks(xticks)
    ax.set_xticklabels(xtick_labels)
    ax.set_yticklabels(xtick_labels)

    bbox = (0., 1.04, 1., .104)
    legend_args = {'bbox_to_anchor': bbox, 'loc': 'upper right'}

    ax.legend(ncol=2, frameon=False, **legend_args, fontsize=14)
    plt.show()

def ts_plot(g, xmin, xmax, x0, ts_length=6, var='x'):
    fig, ax = subplots((7, 5.5))
    ax.set_ylim(xmin, xmax)
    ax.set_xlabel(r'$t$', fontsize=14)
    ax.set_ylabel(r'${}_t$'.format(var), fontsize=14)
    x = np.empty(ts_length)
    x[0] = x0
    for t in range(ts_length-1):
        x[t+1] = g(x[t])
    ax.plot(range(ts_length),
            x,
            'bo-',
            alpha=0.6,
            lw=2,
            label=r'${}_t$'.format(var))
    ax.legend(loc='best', fontsize=14)
    ax.set_xticks(range(ts_length))
    plt.show()
```

Let's create a 45 degree diagram for the Solow model with a fixed set of parameters

```
In [3]: A, s, alpha, delta = 2, 0.3, 0.3, 0.4
```

Here's the update function corresponding to the model.

```
In [4]: def g(k):
            return A * s * k**alpha + (1 - delta) * k
```

Here is the 45 degree plot.

```
In [5]: xmin, xmax = 0, 4  # Suitable plotting region.

        plot45(g, xmin, xmax, 0, num_arrows=0)
```

The plot shows the function g and the 45 degree line.

Think of k_t as a value on the horizontal axis.

To calculate k_{t+1}, we can use the graph of g to see its value on the vertical axis.

Clearly,

- If g lies above the 45 degree line at this point, then we have $k_{t+1} > k_t$.
- If g lies below the 45 degree line at this point, then we have $k_{t+1} < k_t$.
- If g hits the 45 degree line at this point, then we have $k_{t+1} = k_t$, so k_t is a steady state.

For the Solow model, there are two steady states when $S = \mathbb{R}_+ = [0, \infty)$.

- the origin $k = 0$
- the unique positive number such that $k = szk^\alpha + (1 - \delta)k$.

By using some algebra, we can show that in the second case, the steady state is

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

24.4.1 Trajectories

By the preceding discussion, in regions where g lies above the 45 degree line, we know that the trajectory is increasing.

The next figure traces out a trajectory in such a region so we can see this more clearly.

The initial condition is $k_0 = 0.25$.

```
In [6]: k0 = 0.25

        plot45(g, xmin, xmax, k0, num_arrows=5, var='k')
```

We can plot the time series of capital corresponding to the figure above as follows:

```
In [7]: ts_plot(g, xmin, xmax, k0, var='k')
```

Here's a somewhat longer view:

```
In [8]: ts_plot(g, xmin, xmax, k0, ts_length=20, var='k')
```

When capital stock is higher than the unique positive steady state, we see that it declines:

```
In [9]: k0 = 2.95

        plot45(g, xmin, xmax, k0, num_arrows=5, var='k')
```

Here is the time series:

```
In [10]: ts_plot(g, xmin, xmax, k0, var='k')
```

24.4.2 Complex Dynamics

The Solow model is nonlinear but still generates very regular dynamics.

One model that generates irregular dynamics is the **quadratic map**

$$g(x) = 4x(1-x), \qquad x \in [0,1]$$

Let's have a look at the 45 degree diagram.

```
In [11]: xmin, xmax = 0, 1
         g = lambda x: 4 * x * (1 - x)

         x0 = 0.3
         plot45(g, xmin, xmax, x0, num_arrows=0)
```

Now let's look at a typical trajectory.

```
In [12]: plot45(g, xmin, xmax, x0, num_arrows=6)
```

Notice how irregular it is.

Here is the corresponding time series plot.

```
In [13]: ts_plot(g, xmin, xmax, x0, ts_length=6)
```

The irregularity is even clearer over a longer time horizon:

```
In [14]: ts_plot(g, xmin, xmax, x0, ts_length=20)
```

24.5 Exercises

24.5.1 Exercise 1

Consider again the linear model $x_{t+1} = ax_t + b$ with $a \neq 1$.

The unique steady state is $b/(1-a)$.

The steady state is globally stable if $|a| < 1$.

Try to illustrate this graphically by looking at a range of initial conditions.

What differences do you notice in the cases $a \in (-1, 0)$ and $a \in (0, 1)$?

Use $a = 0.5$ and then $a = -0.5$ and study the trajectories

Set $b = 1$ throughout.

24.6 Solutions

24.6.1 Exercise 1

We will start with the case $a = 0.5$.

Let's set up the model and plotting region:

```
In [15]: a, b = 0.5, 1
         xmin, xmax = -1, 3
         g = lambda x: a * x + b
```

Now let's plot a trajectory:

```
In [16]: x0 = -0.5
         plot45(g, xmin, xmax, x0, num_arrows=5)
```

Here is the corresponding time series, which converges towards the steady state.

```
In [17]: ts_plot(g, xmin, xmax, x0, ts_length=10)
```

Now let's try $a = -0.5$ and see what differences we observe.

Let's set up the model and plotting region:

```
In [18]: a, b = -0.5, 1
         xmin, xmax = -1, 3
         g = lambda x: a * x + b
```

Now let's plot a trajectory:

```
In [19]: x0 = -0.5
         plot45(g, xmin, xmax, x0, num_arrows=5)
```

Here is the corresponding time series, which converges towards the steady state.

```
In [20]: ts_plot(g, xmin, xmax, x0, ts_length=10)
```

Once again, we have convergence to the steady state but the nature of convergence differs. In particular, the time series jumps from above the steady state to below it and back again. In the current context, the series is said to exhibit **damped oscillations**.

Chapter 25

AR1 Processes

25.1 Contents

- Overview 25.2
- The AR(1) Model 25.3
- Stationarity and Asymptotic Stability 25.4
- Ergodicity 25.5
- Exercises 25.6
- Solutions 25.7

25.2 Overview

In this lecture we are going to study a very simple class of stochastic models called AR(1) processes.

These simple models are used again and again in economic research to represent the dynamics of series such as

- labor income
- dividends
- productivity, etc.

AR(1) processes can take negative values but are easily converted into positive processes when necessary by a transformation such as exponentiation.

We are going to study AR(1) processes partly because they are useful and partly because they help us understand important concepts.

Let's start with some imports:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

25.3 The AR(1) Model

The **AR(1) model** (autoregressive model of order 1) takes the form

$$X_{t+1} = aX_t + b + cW_{t+1} \tag{1}$$

where a, b, c are scalar-valued parameters.

This law of motion generates a time series $\{X_t\}$ as soon as we specify an initial condition X_0.

This is called the **state process** and the state space is $\mathbb{R}$.

To make things even simpler, we will assume that

- the process $\{W_t\}$ is IID and standard normal,
- the initial condition X_0 is drawn from the normal distribution $N(\mu_0, v_0)$ and
- the initial condition X_0 is independent of $\{W_t\}$.

25.3.1 Moving Average Representation

Iterating backwards from time t, we obtain

$$X_t = aX_{t-1} + b + cW_t = a^2 X_{t-2} + ab + acW_{t-1} + b + cW_t = \cdots$$

If we work all the way back to time zero, we get

$$X_t = a^t X_0 + b \sum_{j=0}^{t-1} a^j + c \sum_{j=0}^{t-1} a^j W_{t-j} \tag{2}$$

Equation (2) shows that X_t is a well defined random variable, the value of which depends on

- the parameters,
- the initial condition X_0 and
- the shocks $W_1, \ldots W_t$ from time $t = 1$ to the present.

Throughout, the symbol ψ_t will be used to refer to the density of this random variable X_t.

25.3.2 Distribution Dynamics

One of the nice things about this model is that it's so easy to trace out the sequence of distributions $\{\psi_t\}$ corresponding to the time series $\{X_t\}$.

To see this, we first note that X_t is normally distributed for each t.

This is immediate form (2), since linear combinations of independent normal random variables are normal.

Given that X_t is normally distributed, we will know the full distribution ψ_t if we can pin down its first two moments.

Let μ_t and v_t denote the mean and variance of X_t respectively.

We can pin down these values from (2) or we can use the following recursive expressions:

$$\mu_{t+1} = a\mu_t + b \quad \text{and} \quad v_{t+1} = a^2 v_t + c^2 \tag{3}$$

These expressions are obtained from (1) by taking, respectively, the expectation and variance of both sides of the equality.

In calculating the second expression, we are using the fact that X_t and W_{t+1} are independent. (This follows from our assumptions and (2).)

Given the dynamics in (2) and initial conditions μ_0, v_0, we obtain μ_t, v_t and hence

$$\psi_t = N(\mu_t, v_t)$$

The following code uses these facts to track the sequence of marginal distributions $\{\psi_t\}$.

The parameters are

```
In [2]: a, b, c = 0.9, 0.1, 0.5

        mu, v = -3.0, 0.6  # initial conditions mu_0, v_0
```

Here's the sequence of distributions:

```
In [3]: from scipy.stats import norm

        sim_length = 10
        grid = np.linspace(-5, 7, 120)

        fig, ax = plt.subplots()

        for t in range(sim_length):
            mu = a * mu + b
            v = a**2 * v + c**2
            ax.plot(grid, norm.pdf(grid, loc=mu, scale=np.sqrt(v)),
                    label=f"$\psi_{t}$",
                    alpha=0.7)

        ax.legend(bbox_to_anchor=[1.05,1],loc=2,borderaxespad=1)

        plt.show()
```

25.4 Stationarity and Asymptotic Stability

Notice that, in the figure above, the sequence $\{\psi_t\}$ seems to be converging to a limiting distribution.

This is even clearer if we project forward further into the future:

```
In [4]: def plot_density_seq(ax, mu_0=-3.0, v_0=0.6, sim_length=60):
            mu, v = mu_0, v_0
            for t in range(sim_length):
                mu = a * mu + b
                v = a**2 * v + c**2
                ax.plot(grid,
                        norm.pdf(grid, loc=mu, scale=np.sqrt(v)),
                        alpha=0.5)

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

Moreover, the limit does not depend on the initial condition.

For example, this alternative density sequence also converges to the same limit.

```
In [5]: fig, ax = plt.subplots()
        plot_density_seq(ax, mu_0=3.0)
        plt.show()
```

In fact it's easy to show that such convergence will occur, regardless of the initial condition, whenever $|a| < 1$.

To see this, we just have to look at the dynamics of the first two moments, as given in (3).

When $|a| < 1$, these sequence converge to the respective limits

$$\mu^* := \frac{b}{1-a} \quad \text{and} \quad v^* = \frac{c^2}{1-a^2} \tag{4}$$

(See our lecture on one dimensional dynamics for background on deterministic convergence.)

Hence

$$\psi_t \to \psi^* = N(\mu^*, v^*) \quad \text{as } t \to \infty \tag{5}$$

We can confirm this is valid for the sequence above using the following code.

In [6]:
```
fig, ax = plt.subplots()
plot_density_seq(ax, mu_0=3.0)

mu_star = b / (1 - a)
std_star = np.sqrt(c**2 / (1 - a**2))  # square root of v_star
psi_star = norm.pdf(grid, loc=mu_star, scale=std_star)
ax.plot(grid, psi_star, 'k-', lw=2, label="$\psi^*$")
ax.legend()

plt.show()
```

As claimed, the sequence $\{\psi_t\}$ converges to ψ^*.

25.4.1 Stationary Distributions

A stationary distribution is a distribution that is a fixed point of the update rule for distributions.

In other words, if ψ_t is stationary, then $\psi_{t+j} = \psi_t$ for all j in $\mathbb{N}$.

A different way to put this, specialized to the current setting, is as follows: a density ψ on $\mathbb{R}$ is **stationary** for the AR(1) process if

$$X_t \sim \psi \quad \Longrightarrow \quad aX_t + b + cW_{t+1} \sim \psi$$

The distribution ψ^* in (5) has this property — checking this is an exercise.

(Of course, we are assuming that $|a| < 1$ so that ψ^* is well defined.)

In fact, it can be shown that no other distribution on $\mathbb{R}$ has this property.

Thus, when $|a| < 1$, the AR(1) model has exactly one stationary density and that density is given by ψ^*.

25.5 Ergodicity

The concept of ergodicity is used in different ways by different authors.

One way to understand it in the present setting is that a version of the Law of Large Numbers is valid for $\{X_t\}$, even though it is not IID.

In particular, averages over time series converge to expectations under the stationary distribution.

Indeed, it can be proved that, whenever $|a| < 1$, we have

$$\frac{1}{m}\sum_{t=1}^{m} h(X_t) \to \int h(x)\psi^*(x)dx \quad \text{as } m \to \infty \tag{6}$$

whenever the integral on the right hand side is finite and well defined.

Notes:

- In (6), convergence holds with probability one.
- The textbook by [117] is a classic reference on ergodicity.

For example, if we consider the identity function $h(x) = x$, we get

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

In other words, the time series sample mean converges to the mean of the stationary distribution.

As will become clear over the next few lectures, ergodicity is a very important concept for statistics and simulation.

25.6 Exercises

25.6.1 Exercise 1

Let k be a natural number.

The k-th central moment of a random variable is defined as

$$M_k := \mathbb{E}[(X - \mathbb{E}X)^k]$$

When that random variable is $N(\mu, \sigma^2)$, it is known that

$$M_k = \begin{cases} 0 & \text{if } k \text{ is odd} \\ \sigma^k (k-1)!! & \text{if } k \text{ is even} \end{cases}$$

Here $n!!$ is the double factorial.

According to (6), we should have, for any $k \in \mathbb{N}$,

$$\frac{1}{m}\sum_{t=1}^{m}(X_t - \mu^*)^k \approx M_k$$

when m is large.

Confirm this by simulation at a range of k using the default parameters from the lecture.

25.6.2 Exercise 2

Write your own version of a one dimensional kernel density estimator, which estimates a density from a sample.

Write it as a class that takes the data X and bandwidth h when initialized and provides a method f such that

$$f(x) = \frac{1}{hn} \sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right)$$

For K use the Gaussian kernel (K is the standard normal density).

Write the class so that the bandwidth defaults to Silverman's rule (see the "rule of thumb" discussion on this page). Test the class you have written by going through the steps

1. simulate data $X_1, \dots, X_n$ from distribution ϕ
2. plot the kernel density estimate over a suitable range
3. plot the density of ϕ on the same figure

for distributions ϕ of the following types

- beta distribution with $\alpha = \beta = 2$
- beta distribution with $\alpha = 2$ and $\beta = 5$
- beta distribution with $\alpha = \beta = 0.5$

Use $n = 500$.

Make a comment on your results. (Do you think this is a good estimator of these distributions?)

25.6.3 Exercise 3

In the lecture we discussed the following fact: for the $AR(1)$ process

$$X_{t+1} = aX_t + b + cW_{t+1}$$

with $\{W_t\}$ iid and standard normal,

$$\psi_t = N(\mu, s^2) \implies \psi_{t+1} = N(a\mu + b, a^2 s^2 + c^2)$$

Confirm this, at least approximately, by simulation. Let

- $a = 0.9$
- $b = 0.0$
- $c = 0.1$
- $\mu = -3$
- $s = 0.2$

First, plot ψ_t and ψ_{t+1} using the true distributions described above.

Second, plot ψ_{t+1} on the same figure (in a different color) as follows:

1. Generate n draws of X_t from the $N(\mu, s^2)$ distribution

2. Update them all using the rule $X_{t+1} = aX_t + b + cW_{t+1}$

3. Use the resulting sample of X_{t+1} values to produce a density estimate via kernel density estimation.

Try this for $n = 2000$ and confirm that the simulation based estimate of ψ_{t+1} does converge to the theoretical distribution.

25.7 Solutions

25.7.1 Exercise 1

In [7]:
```
from numba import njit
from scipy.special import factorial2

@njit
def sample_moments_ar1(k, m=100_000, mu_0=0.0, sigma_0=1.0, seed=1234):
    np.random.seed(seed)
    sample_sum = 0.0
    x = mu_0 + sigma_0 * np.random.randn()
    for t in range(m):
        sample_sum += (x - mu_star)**k
        x = a * x + b + c * np.random.randn()
    return sample_sum / m

def true_moments_ar1(k):
    if k % 2 == 0:
        return std_star**k * factorial2(k - 1)
    else:
        return 0

k_vals = np.arange(6) + 1
sample_moments = np.empty_like(k_vals)
true_moments = np.empty_like(k_vals)

for k_idx, k in enumerate(k_vals):
    sample_moments[k_idx] = sample_moments_ar1(k)
    true_moments[k_idx] = true_moments_ar1(k)

fig, ax = plt.subplots()
ax.plot(k_vals, true_moments, label="true moments")
ax.plot(k_vals, sample_moments, label="sample moments")
ax.legend()

plt.show()
```

25.7.2 Exercise 2

Here is one solution:

```
In [8]: K = norm.pdf

        class KDE:

            def __init__(self, x_data, h=None):

                if h is None:
                    c = x_data.std()
                    n = len(x_data)
                    h = 1.06 * c * n**(-1/5)
                self.h = h
                self.x_data = x_data

            def f(self, x):
                if np.isscalar(x):
                    return K((x - self.x_data) / self.h).mean() * (1/self.h)
                else:
                    y = np.empty_like(x)
                    for i, x_val in enumerate(x):
                       y[i] = K((x_val - self.x_data) / self.h).mean() * (1/self.h)
                    return y

In [9]: def plot_kde(ϕ, x_min=-0.2, x_max=1.2):
            x_data = ϕ.rvs(n)
            kde = KDE(x_data)

            x_grid = np.linspace(-0.2, 1.2, 100)
            fig, ax = plt.subplots()
```

```
    ax.plot(x_grid, kde.f(x_grid), label="estimate")
    ax.plot(x_grid, ϕ.pdf(x_grid), label="true density")
    ax.legend()
    plt.show()
```

```
In [10]: from scipy.stats import beta

         n = 500
         parameter_pairs= (2, 2), (2, 5), (0.5, 0.5)
         for α, β in parameter_pairs:
             plot_kde(beta(α, β))
```

We see that the kernel density estimator is effective when the underlying distribution is smooth but less so otherwise.

25.7.3 Exercise 3

Here is our solution

```
In [11]: a = 0.9
         b = 0.0
         c = 0.1
         μ = -3
         s = 0.2
```

```
In [12]: μ_next = a * μ + b
         s_next = np.sqrt(a**2 * s**2 + c**2)
```

```
In [13]: ψ = lambda x: K((x - μ) / s)
         ψ_next = lambda x: K((x - μ_next) / s_next)
```

```
In [14]: ψ = norm(μ, s)
         ψ_next = norm(μ_next, s_next)
```

```
In [15]: n = 2000
         x_draws = ψ.rvs(n)
         x_draws_next = a * x_draws + b + c * np.random.randn(n)
         kde = KDE(x_draws_next)

         x_grid = np.linspace(μ - 1, μ + 1, 100)
```

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

ax.plot(x_grid, ψ.pdf(x_grid), label="$\psi_t$")
ax.plot(x_grid, ψ_next.pdf(x_grid), label="$\psi_{t+1}$")
ax.plot(x_grid, kde.f(x_grid), label="estimate of $\psi_{t+1}$")

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

The simulated distribution approximately coincides with the theoretical distribution, as predicted.

Chapter 26

Finite Markov Chains

26.1 Contents

- Overview 26.2
- Definitions 26.3
- Simulation 26.4
- Marginal Distributions 26.5
- Irreducibility and Aperiodicity 26.6
- Stationary Distributions 26.7
- Ergodicity 26.8
- Computing Expectations 26.9
- Exercises 26.10
- Solutions 26.11

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

```
In [1]: !pip install --upgrade quantecon
```

26.2 Overview

Markov chains are one of the most useful classes of stochastic processes, being

- simple, flexible and supported by many elegant theoretical results
- valuable for building intuition about random dynamic models
- central to quantitative modeling in their own right

You will find them in many of the workhorse models of economics and finance.

In this lecture, we review some of the theory of Markov chains.

We will also introduce some of the high-quality routines for working with Markov chains available in QuantEcon.py.

Prerequisite knowledge is basic probability and linear algebra.

Let's start with some standard imports:

```
In [2]: import quantecon as qe
        import numpy as np
        from mpl_toolkits.mplot3d import Axes3D
```

```
import matplotlib.pyplot as plt
%matplotlib inline
```

26.3 Definitions

The following concepts are fundamental.

26.3.1 Stochastic Matrices

A **stochastic matrix** (or **Markov matrix**) is an $n \times n$ square matrix P such that

1. each element of P is nonnegative, and
2. each row of P sums to one

Each row of P can be regarded as a probability mass function over n possible outcomes.

It is too not difficult to check Section **??** that if P is a stochastic matrix, then so is the k-th power P^k for all $k \in \mathbb{N}$.

26.3.2 Markov Chains

There is a close connection between stochastic matrices and Markov chains.

To begin, let S be a finite set with n elements $\{x_1, \dots, x_n\}$.

The set S is called the **state space** and $x_1, \dots, x_n$ are the **state values**.

A **Markov chain** $\{X_t\}$ on S is a sequence of random variables on S that have the **Markov property**.

This means that, for any date t and any state $y \in S$,

$$\mathbb{P}\{X_{t+1} = y \,|\, X_t\} = \mathbb{P}\{X_{t+1} = y \,|\, X_t, X_{t-1}, \dots\} \tag{1}$$

In other words, knowing the current state is enough to know probabilities for future states.

In particular, the dynamics of a Markov chain are fully determined by the set of values

$$P(x, y) := \mathbb{P}\{X_{t+1} = y \,|\, X_t = x\} \qquad (x, y \in S) \tag{2}$$

By construction,

- $P(x, y)$ is the probability of going from x to y in one unit of time (one step)
- $P(x, \cdot)$ is the conditional distribution of X_{t+1} given $X_t = x$

We can view P as a stochastic matrix where

$$P_{ij} = P(x_i, x_j) \qquad 1 \le i, j \le n$$

Going the other way, if we take a stochastic matrix P, we can generate a Markov chain $\{X_t\}$ as follows:

- draw X_0 from some specified distribution
- for each $t = 0, 1, ...$, draw X_{t+1} from $P(X_t, \cdot)$

By construction, the resulting process satisfies (2).

26.3.3 Example 1

Consider a worker who, at any given time t, is either unemployed (state 0) or employed (state 1).

Suppose that, over a one month period,

1. An unemployed worker finds a job with probability $\alpha \in (0, 1)$.
2. An employed worker loses her job and becomes unemployed with probability $\beta \in (0, 1)$.

In terms of a Markov model, we have

- $S = \{0, 1\}$
- $P(0, 1) = \alpha$ and $P(1, 0) = \beta$

We can write out the transition probabilities in matrix form as

$$P = \begin{pmatrix} 1 - \alpha & \alpha \\ \beta & 1 - \beta \end{pmatrix} \tag{3}$$

Once we have the values α and β, we can address a range of questions, such as

- What is the average duration of unemployment?
- Over the long-run, what fraction of time does a worker find herself unemployed?
- Conditional on employment, what is the probability of becoming unemployed at least once over the next 12 months?

We'll cover such applications below.

26.3.4 Example 2

Using US unemployment data, Hamilton [70] estimated the stochastic matrix

$$P = \begin{pmatrix} 0.971 & 0.029 & 0 \\ 0.145 & 0.778 & 0.077 \\ 0 & 0.508 & 0.492 \end{pmatrix}$$

where

- the frequency is monthly
- the first state represents "normal growth"
- the second state represents "mild recession"
- the third state represents "severe recession"

For example, the matrix tells us that when the state is normal growth, the state will again be normal growth next month with probability 0.97.

In general, large values on the main diagonal indicate persistence in the process $\{X_t\}$.

This Markov process can also be represented as a directed graph, with edges labeled by transition probabilities

Here "ng" is normal growth, "mr" is mild recession, etc.

26.4 Simulation

One natural way to answer questions about Markov chains is to simulate them.

(To approximate the probability of event E, we can simulate many times and count the fraction of times that E occurs).

Nice functionality for simulating Markov chains exists in QuantEcon.py.

- Efficient, bundled with lots of other useful routines for handling Markov chains.

However, it's also a good exercise to roll our own routines — let's do that first and then come back to the methods in QuantEcon.py.

In these exercises, we'll take the state space to be $S = 0, \dots, n-1$.

26.4.1 Rolling Our Own

To simulate a Markov chain, we need its stochastic matrix P and a probability distribution ψ for the initial state to be drawn from.

The Markov chain is then constructed as discussed above. To repeat:

1. At time $t = 0$, the X_0 is chosen from ψ.
2. At each subsequent time t, the new state X_{t+1} is drawn from $P(X_t, \cdot)$.

To implement this simulation procedure, we need a method for generating draws from a discrete distribution.

For this task, we'll use `random.draw` from QuantEcon, which works as follows:

```
In [3]: ψ = (0.3, 0.7)           # probabilities over {0, 1}
        cdf = np.cumsum(ψ)       # convert into cummulative distribution
        qe.random.draw(cdf, 5)   # generate 5 independent draws from ψ

Out[3]: array([1, 1, 0, 1, 0])
```

We'll write our code as a function that takes the following three arguments

- A stochastic matrix `P`
- An initial state `init`
- A positive integer `sample_size` representing the length of the time series the function should return

```
In [4]: def mc_sample_path(P, ψ_0=None, sample_size=1_000):

            # set up
            P = np.asarray(P)
            X = np.empty(sample_size, dtype=int)

            # Convert each row of P into a cdf
            n = len(P)
            P_dist = [np.cumsum(P[i, :]) for i in range(n)]

            # draw initial state, defaulting to 0
            if ψ_0 is not None:
                X_0 = qe.random.draw(np.cumsum(ψ_0))
            else:
                X_0 = 0

            # simulate
            X[0] = X_0
            for t in range(sample_size - 1):
                X[t+1] = qe.random.draw(P_dist[X[t]])

            return X
```

Let's see how it works using the small matrix

```
In [5]: P = [[0.4, 0.6],
             [0.2, 0.8]]
```

As we'll see later, for a long series drawn from `P`, the fraction of the sample that takes value 0 will be about 0.25.

Moreover, this is true, regardless of the initial distribution from with X_0 is drawn.

The following code illustrates this

```
In [6]: X = mc_sample_path(P, ψ_0=[0.1, 0.9], sample_size=100_000)
        np.mean(X == 0)

Out[6]: 0.24663
```

You can try changing the initial distribution to confirm that the output is always close to 0.25.

26.4.2 Using QuantEcon's Routines

As discussed above, QuantEcon.py has routines for handling Markov chains, including simulation.

Here's an illustration using the same P as the preceding example

In [7]:
```
from quantecon import MarkovChain

mc = qe.MarkovChain(P)
X = mc.simulate(ts_length=1_000_000)
np.mean(X == 0)
```

Out[7]:
```
0.249979
```

The QuantEcon.py routine is JIT compiled and much faster.

In [8]:
```
%time mc_sample_path(P, sample_size=1_000_000) # Our version
```

```
    CPU times: user 1.22 s, sys: 7.54 ms, total: 1.22 s
Wall time: 1.26 s
```

Out[8]:
```
array([0, 0, 0, …, 1, 1, 1])
```

In [9]:
```
%time mc.simulate(ts_length=1_000_000) # qe version
```

```
    CPU times: user 37.2 ms, sys: 11.5 ms, total: 48.7 ms
Wall time: 51.3 ms
```

Out[9]:
```
array([0, 0, 0, …, 1, 1, 1])
```

Adding State Values and Initial Conditions

If we wish to, we can provide a specification of state values to `MarkovChain`.

These state values can be integers, floats, or even strings.

The following code illustrates

In [10]:
```
mc = qe.MarkovChain(P, state_values=('unemployed', 'employed'))
mc.simulate(ts_length=4, init='employed')
```

Out[10]:
```
array(['employed', 'employed', 'unemployed', 'unemployed'], dtype='<U10')
```

In [11]:
```
mc.simulate(ts_length=4, init='unemployed')
```

Out[11]:
```
array(['unemployed', 'unemployed', 'unemployed', 'unemployed'],
      dtype='<U10')
```

In [12]:
```
mc.simulate(ts_length=4)  # Start at randomly chosen initial state
```

Out[12]:
```
array(['unemployed', 'employed', 'employed', 'employed'], dtype='<U10')
```

If we want to simulate with output as indices rather than state values we can use

In [13]:
```
mc.simulate_indices(ts_length=4)
```

Out[13]:
```
array([1, 1, 1, 1])
```

26.5 Marginal Distributions

Suppose that

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

What then is the distribution of X_{t+1}, or, more generally, of X_{t+m}?

To answer this, we let ψ_t be the distribution of X_t for $t = 0, 1, 2, \ldots$.

Our first aim is to find ψ_{t+1} given ψ_t and P.

To begin, pick any $y \in S$.

Using the law of total probability, we can decompose the probability that $X_{t+1} = y$ as follows:

$$\mathbb{P}\{X_{t+1} = y\} = \sum_{x \in S} \mathbb{P}\{X_{t+1} = y \,|\, X_t = x\} \cdot \mathbb{P}\{X_t = x\}$$

In words, to get the probability of being at y tomorrow, we account for all ways this can happen and sum their probabilities.

Rewriting this statement in terms of marginal and conditional probabilities gives

$$\psi_{t+1}(y) = \sum_{x \in S} P(x, y)\psi_t(x)$$

There are n such equations, one for each $y \in S$.

If we think of ψ_{t+1} and ψ_t as *row vectors* (as is traditional in this literature), these n equations are summarized by the matrix expression

$$\psi_{t+1} = \psi_t P \tag{4}$$

In other words, to move the distribution forward one unit of time, we postmultiply by P.

By repeating this m times we move forward m steps into the future.

Hence, iterating on (4), the expression $\psi_{t+m} = \psi_t P^m$ is also valid — here P^m is the m-th power of P.

As a special case, we see that if ψ_0 is the initial distribution from which X_0 is drawn, then $\psi_0 P^m$ is the distribution of X_m.

This is very important, so let's repeat it

$$X_0 \sim \psi_0 \quad \Longrightarrow \quad X_m \sim \psi_0 P^m \tag{5}$$

and, more generally,

$$X_t \sim \psi_t \quad \Longrightarrow \quad X_{t+m} \sim \psi_t P^m \tag{6}$$

26.5.1 Multiple Step Transition Probabilities

We know that the probability of transitioning from x to y in one step is $P(x, y)$.

It turns out that the probability of transitioning from x to y in m steps is $P^m(x, y)$, the (x, y)-th element of the m-th power of P.

To see why, consider again (6), but now with ψ_t putting all probability on state x

- 1 in the x-th position and zero elsewhere

Inserting this into (6), we see that, conditional on $X_t = x$, the distribution of X_{t+m} is the x-th row of P^m.

In particular

$$\mathbb{P}\{X_{t+m} = y \,|\, X_t = x\} = P^m(x, y) = (x, y)\text{-th element of } P^m$$

26.5.2 Example: Probability of Recession

Recall the stochastic matrix P for recession and growth considered above.

Suppose that the current state is unknown — perhaps statistics are available only at the *end* of the current month.

We estimate the probability that the economy is in state x to be $\psi(x)$.

The probability of being in recession (either mild or severe) in 6 months time is given by the inner product

$$\psi P^6 \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

26.5.3 Example 2: Cross-Sectional Distributions

The marginal distributions we have been studying can be viewed either as probabilities or as cross-sectional frequencies in large samples.

To illustrate, recall our model of employment/unemployment dynamics for a given worker discussed above.

Consider a large population of workers, each of whose lifetime experience is described by the specified dynamics, independent of one another.

Let ψ be the current *cross-sectional* distribution over $\{0, 1\}$.

The cross-sectional distribution records the fractions of workers employed and unemployed at a given moment.

- For example, $\psi(0)$ is the unemployment rate.

What will the cross-sectional distribution be in 10 periods hence?

The answer is ψP^{10}, where P is the stochastic matrix in (3).

This is because each worker is updated according to P, so ψP^{10} represents probabilities for a single randomly selected worker.

But when the sample is large, outcomes and probabilities are roughly equal (by the Law of Large Numbers).

So for a very large (tending to infinite) population, ψP^{10} also represents the fraction of workers in each state.

This is exactly the cross-sectional distribution.

26.6 Irreducibility and Aperiodicity

Irreducibility and aperiodicity are central concepts of modern Markov chain theory.

Let's see what they're about.

26.6.1 Irreducibility

Let P be a fixed stochastic matrix.

Two states x and y are said to **communicate** with each other if there exist positive integers j and k such that

$$P^j(x, y) > 0 \quad \text{and} \quad P^k(y, x) > 0$$

In view of our discussion above, this means precisely that

- state x can be reached eventually from state y, and
- state y can be reached eventually from state x

The stochastic matrix P is called **irreducible** if all states communicate; that is, if x and y communicate for all (x, y) in $S \times S$.

For example, consider the following transition probabilities for wealth of a fictitious set of households

We can translate this into a stochastic matrix, putting zeros where there's no edge between nodes

$$P := \begin{pmatrix} 0.9 & 0.1 & 0 \\ 0.4 & 0.4 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{pmatrix}$$

It's clear from the graph that this stochastic matrix is irreducible: we can reach any state from any other state eventually.

We can also test this using QuantEcon.py's MarkovChain class

```
In [14]: P = [[0.9, 0.1, 0.0],
              [0.4, 0.4, 0.2],
              [0.1, 0.1, 0.8]]

         mc = qe.MarkovChain(P, ('poor', 'middle', 'rich'))
         mc.is_irreducible
```

```
Out[14]: True
```

Here's a more pessimistic scenario, where the poor are poor forever

This stochastic matrix is not irreducible, since, for example, rich is not accessible from poor.

Let's confirm this

```
In [15]: P = [[1.0, 0.0, 0.0],
              [0.1, 0.8, 0.1],
              [0.0, 0.2, 0.8]]

         mc = qe.MarkovChain(P, ('poor', 'middle', 'rich'))
         mc.is_irreducible
```

```
Out[15]: False
```

We can also determine the "communication classes"

```
In [16]: mc.communication_classes
```

```
Out[16]: [array(['poor'], dtype='<U6'), array(['middle', 'rich'], dtype='<U6')]
```

It might be clear to you already that irreducibility is going to be important in terms of long run outcomes.

For example, poverty is a life sentence in the second graph but not the first.

We'll come back to this a bit later.

26.6.2 Aperiodicity

Loosely speaking, a Markov chain is called periodic if it cycles in a predictible way, and aperiodic otherwise.

Here's a trivial example with three states

The chain cycles with period 3:

```
In [17]: P = [[0, 1, 0],
              [0, 0, 1],
              [1, 0, 0]]

         mc = qe.MarkovChain(P)
         mc.period
```

```
Out[17]: 3
```

More formally, the **period** of a state x is the greatest common divisor of the set of integers

$$D(x) := \{j \geq 1 : P^j(x, x) > 0\}$$

In the last example, $D(x) = \{3, 6, 9, ...\}$ for every state x, so the period is 3.

A stochastic matrix is called **aperiodic** if the period of every state is 1, and **periodic** otherwise.

For example, the stochastic matrix associated with the transition probabilities below is periodic because, for example, state a has period 2

We can confirm that the stochastic matrix is periodic as follows

```
In [18]: P = [[0.0, 1.0, 0.0, 0.0],
              [0.5, 0.0, 0.5, 0.0],
              [0.0, 0.5, 0.0, 0.5],
              [0.0, 0.0, 1.0, 0.0]]

         mc = qe.MarkovChain(P)
         mc.period
```

Out[18]: 2

In [19]:
```
mc.is_aperiodic
```

Out[19]: False

26.7 Stationary Distributions

As seen in (4), we can shift probabilities forward one unit of time via postmultiplication by P.

Some distributions are invariant under this updating process — for example,

In [20]:
```
P = np.array([[0.4, 0.6],
              [0.2, 0.8]])
ψ = (0.25, 0.75)
ψ @ P
```

Out[20]:
```
array([0.25, 0.75])
```

Such distributions are called **stationary**, or **invariant**.

Formally, a distribution ψ^* on S is called **stationary** for P if $\psi^* = \psi^* P$.

(This is the same notion of stationarity that we learned about in the lecture on AR(1) processes applied to a different setting.)

From this equality, we immediately get $\psi^* = \psi^* P^t$ for all t.

This tells us an important fact: If the distribution of X_0 is a stationary distribution, then X_t will have this same distribution for all t.

Hence stationary distributions have a natural interpretation as stochastic steady states — we'll discuss this more in just a moment.

Mathematically, a stationary distribution is a fixed point of P when P is thought of as the map $\psi \mapsto \psi P$ from (row) vectors to (row) vectors.

Theorem. Every stochastic matrix P has at least one stationary distribution.

(We are assuming here that the state space S is finite; if not more assumptions are required)

For proof of this result, you can apply Brouwer's fixed point theorem, or see EDTC, theorem 4.3.5.

There may in fact be many stationary distributions corresponding to a given stochastic matrix P.

- For example, if P is the identity matrix, then all distributions are stationary.

Since stationary distributions are long run equilibria, to get uniqueness we require that initial conditions are not infinitely persistent.

Infinite persistence of initial conditions occurs if certain regions of the state space cannot be accessed from other regions, which is the opposite of irreducibility.

This gives some intuition for the following fundamental theorem.

Theorem. If P is both aperiodic and irreducible, then

1. P has exactly one stationary distribution ψ^*.
2. For any initial distribution ψ_0, we have $\|\psi_0 P^t - \psi^*\| \to 0$ as $t \to \infty$.

For a proof, see, for example, theorem 5.2 of [65].

(Note that part 1 of the theorem requires only irreducibility, whereas part 2 requires both irreducibility and aperiodicity)

A stochastic matrix satisfying the conditions of the theorem is sometimes called **uniformly ergodic**.

One easy sufficient condition for aperiodicity and irreducibility is that every element of P is strictly positive.

- Try to convince yourself of this.

26.7.1 Example

Recall our model of employment/unemployment dynamics for a given worker discussed above.

Assuming $\alpha \in (0, 1)$ and $\beta \in (0, 1)$, the uniform ergodicity condition is satisfied.

Let $\psi^* = (p, 1 - p)$ be the stationary distribution, so that p corresponds to unemployment (state 0).

Using $\psi^* = \psi^* P$ and a bit of algebra yields

$$p = \frac{\beta}{\alpha + \beta}$$

This is, in some sense, a steady state probability of unemployment — more on interpretation below.

Not surprisingly it tends to zero as $\beta \to 0$, and to one as $\alpha \to 0$.

26.7.2 Calculating Stationary Distributions

As discussed above, a given Markov matrix P can have many stationary distributions.

That is, there can be many row vectors ψ such that $\psi = \psi P$.

In fact if P has two distinct stationary distributions ψ_1, ψ_2 then it has infinitely many, since in this case, as you can verify,

$$\psi_3 := \lambda \psi_1 + (1 - \lambda) \psi_2$$

is a stationary distribution for P for any $\lambda \in [0, 1]$.

If we restrict attention to the case where only one stationary distribution exists, one option for finding it is to try to solve the linear system $\psi(I_n - P) = 0$ for ψ, where I_n is the $n \times n$ identity.

But the zero vector solves this equation.

Hence we need to impose the restriction that the solution must be a probability distribution.

A suitable algorithm is implemented in QuantEcon.py — the next code block illustrates

In [21]:
```
P = [[0.4, 0.6],
     [0.2, 0.8]]

mc = qe.MarkovChain(P)
mc.stationary_distributions  # Show all stationary distributions
```

Out[21]:
```
array([[0.25, 0.75]])
```

The stationary distribution is unique.

26.7.3 Convergence to Stationarity

Part 2 of the Markov chain convergence theorem stated above tells us that the distribution of X_t converges to the stationary distribution regardless of where we start off.

This adds considerable weight to our interpretation of ψ^* as a stochastic steady state.

The convergence in the theorem is illustrated in the next figure

In [22]:
```
P = ((0.971, 0.029, 0.000),
     (0.145, 0.778, 0.077),
     (0.000, 0.508, 0.492))
P = np.array(P)

ψ = (0.0, 0.2, 0.8)        # Initial condition

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

ax.set(xlim=(0, 1), ylim=(0, 1), zlim=(0, 1),
       xticks=(0.25, 0.5, 0.75),
       yticks=(0.25, 0.5, 0.75),
       zticks=(0.25, 0.5, 0.75))

x_vals, y_vals, z_vals = [], [], []
for t in range(20):
    x_vals.append(ψ[0])
    y_vals.append(ψ[1])
    z_vals.append(ψ[2])
    ψ = ψ @ P

ax.scatter(x_vals, y_vals, z_vals, c='r', s=60)
ax.view_init(30, 210)

mc = qe.MarkovChain(P)
ψ_star = mc.stationary_distributions[0]
ax.scatter(ψ_star[0], ψ_star[1], ψ_star[2], c='k', s=60)

plt.show()
```

Here

- P is the stochastic matrix for recession and growth considered above.
- The highest red dot is an arbitrarily chosen initial probability distribution ψ, represented as a vector in $\mathbb{R}^3$.
- The other red dots are the distributions ψP^t for $t = 1, 2, \ldots$.
- The black dot is ψ^*.

You might like to try experimenting with different initial conditions.

26.8 Ergodicity

Under irreducibility, yet another important result obtains: For all $x \in S$,

$$\frac{1}{m} \sum_{t=1}^{m} \mathbf{1}\{X_t = x\} \to \psi^*(x) \quad \text{as } m \to \infty \tag{7}$$

Here

- $\mathbf{1}\{X_t = x\} = 1$ if $X_t = x$ and zero otherwise
- convergence is with probability one
- the result does not depend on the distribution (or value) of X_0

The result tells us that the fraction of time the chain spends at state x converges to $\psi^*(x)$ as time goes to infinity.

This gives us another way to interpret the stationary distribution — provided that the convergence result in (7) is valid.

The convergence in (7) is a special case of a law of large numbers result for Markov chains — see EDTC, section 4.3.4 for some additional information.

26.8.1 Example

Recall our cross-sectional interpretation of the employment/unemployment model discussed above.

Assume that $\alpha \in (0,1)$ and $\beta \in (0,1)$, so that irreducibility and aperiodicity both hold.

We saw that the stationary distribution is $(p, 1-p)$, where

$$p = \frac{\beta}{\alpha + \beta}$$

In the cross-sectional interpretation, this is the fraction of people unemployed.

In view of our latest (ergodicity) result, it is also the fraction of time that a worker can expect to spend unemployed.

Thus, in the long-run, cross-sectional averages for a population and time-series averages for a given person coincide.

This is one interpretation of the notion of ergodicity.

26.9 Computing Expectations

We are interested in computing expectations of the form

$$\mathbb{E}[h(X_t)] \tag{8}$$

and conditional expectations such as

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] \tag{9}$$

where

- $\{X_t\}$ is a Markov chain generated by $n \times n$ stochastic matrix P
- h is a given function, which, in expressions involving matrix algebra, we'll think of as the column vector

$$h = \begin{pmatrix} h(x_1) \\ \vdots \\ h(x_n) \end{pmatrix}$$

The unconditional expectation (8) is easy: We just sum over the distribution of X_t to get

$$\mathbb{E}[h(X_t)] = \sum_{x \in S} (\psi P^t)(x) h(x)$$

Here ψ is the distribution of X_0.

Since ψ and hence ψP^t are row vectors, we can also write this as

$$\mathbb{E}[h(X_t)] = \psi P^t h$$

For the conditional expectation (9), we need to sum over the conditional distribution of X_{t+k} given $X_t = x$.

We already know that this is $P^k(x, \cdot)$, so

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] = (P^k h)(x) \tag{10}$$

The vector $P^k h$ stores the conditional expectation $\mathbb{E}[h(X_{t+k}) \mid X_t = x]$ over all x.

26.9.1 Expectations of Geometric Sums

Sometimes we also want to compute expectations of a geometric sum, such as $\sum_t \beta^t h(X_t)$.

In view of the preceding discussion, this is

$$\mathbb{E}\left[\sum_{j=0}^{\infty} \beta^j h(X_{t+j}) \mid X_t = x\right] = [(I - \beta P)^{-1} h](x)$$

where

$$(I - \beta P)^{-1} = I + \beta P + \beta^2 P^2 + \cdots$$

Premultiplication by $(I - \beta P)^{-1}$ amounts to "applying the **resolvent operator**".

26.10 Exercises

26.10.1 Exercise 1

According to the discussion above, if a worker's employment dynamics obey the stochastic matrix

$$P = \begin{pmatrix} 1 - \alpha & \alpha \\ \beta & 1 - \beta \end{pmatrix}$$

with $\alpha \in (0, 1)$ and $\beta \in (0, 1)$, then, in the long-run, the fraction of time spent unemployed will be

$$p := \frac{\beta}{\alpha + \beta}$$

In other words, if $\{X_t\}$ represents the Markov chain for employment, then $\bar{X}_m \to p$ as $m \to \infty$, where

$$\bar{X}_m := \frac{1}{m} \sum_{t=1}^{m} \mathbf{1}\{X_t = 0\}$$

The exercise is to illustrate this convergence by computing $\bar{X}_m$ for large m and checking that it is close to p.

You will see that this statement is true regardless of the choice of initial condition or the values of α, β, provided both lie in $(0, 1)$.

26.10.2 Exercise 2

A topic of interest for economics and many other disciplines is *ranking*.

Let's now consider one of the most practical and important ranking problems — the rank assigned to web pages by search engines.

(Although the problem is motivated from outside of economics, there is in fact a deep connection between search ranking systems and prices in certain competitive equilibria — see [48])

To understand the issue, consider the set of results returned by a query to a web search engine.

For the user, it is desirable to

1. receive a large set of accurate matches
2. have the matches returned in order, where the order corresponds to some measure of "importance"

Ranking according to a measure of importance is the problem we now consider.

The methodology developed to solve this problem by Google founders Larry Page and Sergey Brin is known as PageRank.

To illustrate the idea, consider the following diagram

Imagine that this is a miniature version of the WWW, with

- each node representing a web page
- each arrow representing the existence of a link from one page to another

Now let's think about which pages are likely to be important, in the sense of being valuable to a search engine user.

One possible criterion for the importance of a page is the number of inbound links — an indication of popularity.

By this measure, `m` and `j` are the most important pages, with 5 inbound links each.

However, what if the pages linking to `m`, say, are not themselves important?

Thinking this way, it seems appropriate to weight the inbound nodes by relative importance.

The PageRank algorithm does precisely this.

A slightly simplified presentation that captures the basic idea is as follows.

Letting j be (the integer index of) a typical page and r_j be its ranking, we set

$$r_j = \sum_{i \in L_j} \frac{r_i}{\ell_i}$$

where

- ℓ_i is the total number of outbound links from i
- L_j is the set of all pages i such that i has a link to j

This is a measure of the number of inbound links, weighted by their own ranking (and normalized by $1/\ell_i$).

There is, however, another interpretation, and it brings us back to Markov chains.

Let P be the matrix given by $P(i,j) = \mathbf{1}\{i \to j\}/\ell_i$ where $\mathbf{1}\{i \to j\} = 1$ if i has a link to j and zero otherwise.

The matrix P is a stochastic matrix provided that each page has at least one link.

With this definition of P we have

$$r_j = \sum_{i \in L_j} \frac{r_i}{\ell_i} = \sum_{\text{all } i} \mathbf{1}\{i \to j\} \frac{r_i}{\ell_i} = \sum_{\text{all } i} P(i,j) r_i$$

Writing r for the row vector of rankings, this becomes $r = rP$.

Hence r is the stationary distribution of the stochastic matrix P.

Let's think of $P(i,j)$ as the probability of "moving" from page i to page j.

The value $P(i,j)$ has the interpretation

- $P(i,j) = 1/k$ if i has k outbound links and j is one of them
- $P(i,j) = 0$ if i has no direct link to j

Thus, motion from page to page is that of a web surfer who moves from one page to another by randomly clicking on one of the links on that page.

Here "random" means that each link is selected with equal probability.

Since r is the stationary distribution of P, assuming that the uniform ergodicity condition is valid, we can interpret r_j as the fraction of time that a (very persistent) random surfer spends at page j.

Your exercise is to apply this ranking algorithm to the graph pictured above and return the

list of pages ordered by rank.

There is a total of 14 nodes (i.e., web pages), the first named `a` and the last named `n`.

A typical line from the file has the form

```
d -> h;
```

This should be interpreted as meaning that there exists a link from `d` to `h`.

The data for this graph is shown below, and read into a file called `web_graph_data.txt` when the cell is executed.

In [23]:
```
%%file web_graph_data.txt
a -> d;
a -> f;
b -> j;
b -> k;
b -> m;
c -> c;
c -> g;
c -> j;
c -> m;
d -> f;
d -> h;
d -> k;
e -> d;
e -> h;
e -> l;
f -> a;
f -> b;
f -> j;
f -> l;
g -> b;
g -> j;
h -> d;
h -> g;
h -> l;
h -> m;
i -> g;
i -> h;
i -> n;
j -> e;
j -> i;
j -> k;
k -> n;
l -> m;
m -> g;
n -> c;
n -> j;
n -> m;
```

```
Overwriting web_graph_data.txt
```

To parse this file and extract the relevant information, you can use regular expressions.

The following code snippet provides a hint as to how you can go about this

```
In [24]: import re
         re.findall('\w', 'x +++ y ****** z')  # \w matches alphanumerics

Out[24]: ['x', 'y', 'z']

In [25]: re.findall('\w', 'a ^^ b &&& $$ c')

Out[25]: ['a', 'b', 'c']
```

When you solve for the ranking, you will find that the highest ranked node is in fact `g`, while the lowest is `a`.

26.10.3 Exercise 3

In numerical work, it is sometimes convenient to replace a continuous model with a discrete one.

In particular, Markov chains are routinely generated as discrete approximations to AR(1) processes of the form

$$y_{t+1} = \rho y_t + u_{t+1}$$

Here u_t is assumed to be IID and $N(0, \sigma_u^2)$.

The variance of the stationary probability distribution of $\{y_t\}$ is

$$\sigma_y^2 := \frac{\sigma_u^2}{1 - \rho^2}$$

Tauchen's method [156] is the most common method for approximating this continuous state process with a finite state Markov chain.

A routine for this already exists in QuantEcon.py but let's write our own version as an exercise.

As a first step, we choose

- n, the number of states for the discrete approximation
- m, an integer that parameterizes the width of the state space

Next, we create a state space $\{x_0, \dots, x_{n-1}\} \subset \mathbb{R}$ and a stochastic $n \times n$ matrix P such that

- $x_0 = -m\,\sigma_y$
- $x_{n-1} = m\,\sigma_y$
- $x_{i+1} = x_i + s$ where $s = (x_{n-1} - x_0)/(n-1)$

Let F be the cumulative distribution function of the normal distribution $N(0, \sigma_u^2)$.

The values $P(x_i, x_j)$ are computed to approximate the AR(1) process — omitting the derivation, the rules are as follows:

1. If $j = 0$, then set

$$P(x_i, x_j) = P(x_i, x_0) = F(x_0 - \rho x_i + s/2)$$

1. If $j = n - 1$, then set

$$P(x_i, x_j) = P(x_i, x_{n-1}) = 1 - F(x_{n-1} - \rho x_i - s/2)$$

1. Otherwise, set

$$P(x_i, x_j) = F(x_j - \rho x_i + s/2) - F(x_j - \rho x_i - s/2)$$

The exercise is to write a function `approx_markov(rho, sigma_u, m=3, n=7)` that returns $\{x_0, \ldots, x_{n-1}\} \subset \mathbb{R}$ and $n \times n$ matrix P as described above.

- Even better, write a function that returns an instance of QuantEcon.py's MarkovChain class.

26.11 Solutions

26.11.1 Exercise 1

We will address this exercise graphically.

The plots show the time series of $\bar{X}_m - p$ for two initial conditions.

As m gets large, both series converge to zero.

In [26]:
```
α = β = 0.1
N = 10000
p = β / (α + β)

P = ((1 - α,        α),                # Careful: P and p are distinct
     (    β,    1 - β))
P = np.array(P)
mc = MarkovChain(P)

fig, ax = plt.subplots(figsize=(9, 6))
ax.set_ylim(-0.25, 0.25)
ax.grid()
ax.hlines(0, 0, N, lw=2, alpha=0.6)   # Horizonal line at zero

for x0, col in ((0, 'blue'), (1, 'green')):
    # Generate time series for worker that starts at x0
    X = mc.simulate(N, init=x0)
    # Compute fraction of time spent unemployed, for each n
    X_bar = (X == 0).cumsum() / (1 + np.arange(N, dtype=float))
    # Plot
   ax.fill_between(range(N), np.zeros(N), X_bar - p, color=col, alpha=0.1)
    ax.plot(X_bar - p, color=col, label=f'$X_0 = \, {x0} $')
    # Overlay in black--make lines clearer
    ax.plot(X_bar - p, 'k-', alpha=0.6)

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

26.11.2 Exercise 2

In [27]:
```
"""
Return list of pages, ordered by rank
"""
import re
from operator import itemgetter

infile = 'web_graph_data.txt'
alphabet = 'abcdefghijklmnopqrstuvwxyz'

n = 14 # Total number of web pages (nodes)

# Create a matrix Q indicating existence of links
#  * Q[i, j] = 1 if there is a link from i to j
#  * Q[i, j] = 0 otherwise
Q = np.zeros((n, n), dtype=int)
f = open(infile, 'r')
edges = f.readlines()
f.close()
for edge in edges:
    from_node, to_node = re.findall('\w', edge)
    i, j = alphabet.index(from_node), alphabet.index(to_node)
    Q[i, j] = 1
# Create the corresponding Markov matrix P
P = np.empty((n, n))
for i in range(n):
    P[i, :] = Q[i, :] / Q[i, :].sum()
mc = MarkovChain(P)
# Compute the stationary distribution r
r = mc.stationary_distributions[0]
```

```
    ranked_pages = {alphabet[i] : r[i] for i in range(n)}
    # Print solution, sorted from highest to lowest rank
    print('Rankings\n ***')
    for name, rank in sorted(ranked_pages.items(), key=itemgetter(1),
↪reverse=1):
        print(f'{name}: {rank:.4}')
```

```
    Rankings
 ***
g: 0.1607
j: 0.1594
m: 0.1195
n: 0.1088
k: 0.09106
b: 0.08326
e: 0.05312
i: 0.05312
c: 0.04834
h: 0.0456
l: 0.03202
d: 0.03056
f: 0.01164
a: 0.002911
```

26.11.3 Exercise 3

A solution from the QuantEcon.py library can be found here.

Footnotes

[1] Hint: First show that if P and Q are stochastic matrices then so is their product — to check the row sums, try post multiplying by a column vector of ones. Finally, argue that P^n is a stochastic matrix using induction.

Chapter 27

Inventory Dynamics

27.1 Contents

- Overview 27.2
- Sample Paths 27.3
- Marginal Distributions 27.4
- Exercises 27.5
- Solutions 27.6

27.2 Overview

In this lecture we will study the time path of inventories for firms that follow so-called s-S inventory dynamics.

Such firms

1. wait until inventory falls below some level s and then
2. order sufficent quantities to bring their inventory back up to capacity S.

These kinds of policies are common in practice and also optimal in certain circumstances.

A review of early literature and some macroeconomic implications can be found in [30].

Here our main aim is to learn more about simulation, time series and Markov dynamics.

While our Markov environment and many of the concepts we consider are related to those found in our lecture on finite Markov chains, the state space is a continuum in the current application.

Let's start with some imports

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

        from numba import njit, jitclass, float64, prange
```

27.3 Sample Paths

Consider a firm with inventory X_t.

The firm waits until $X_t \leq s$ and then restocks up to S units.

It faces stochastic demand $\{D_t\}$, which we assume is IID.

With notation $a^+ := \max\{a, 0\}$, inventory dynamics can be written as

$$X_{t+1} = \begin{cases} (S - D_{t+1})^+ & \text{if } X_t \leq s \\ (X_t - D_{t+1})^+ & \text{if } X_t > s \end{cases}$$

In what follows, we will assume that each D_t is lognormal, so that

$$D_t = \exp(\mu + \sigma Z_t)$$

where μ and σ are parameters and $\{Z_t\}$ is IID and standard normal.

Here's a class that stores parameters and generates time paths for inventory.

In [2]:
```
firm_data = [
    ('s', float64),          # restock trigger level
    ('S', float64),          # capacity
    ('mu', float64),         # shock location parameter
    ('sigma', float64)       # shock scale parameter
]


@jitclass(firm_data)
class Firm:

    def __init__(self, s=10, S=100, mu=1.0, sigma=0.5):

        self.s, self.S, self.mu, self.sigma = s, S, mu, sigma

    def update(self, x):
        "Update the state from t to t+1 given current state x."

        Z = np.random.randn()
        D = np.exp(self.mu + self.sigma * Z)
        if x <= self.s:
            return max(self.S - D, 0)
        else:
            return max(x - D, 0)

    def sim_inventory_path(self, x_init, sim_length):

        X = np.empty(sim_length)
        X[0] = x_init

        for t in range(sim_length-1):
            X[t+1] = self.update(X[t])
        return X
```

Let's run a first simulation, of a single path:

In [3]:
```
firm = Firm()

s, S = firm.s, firm.S
sim_length = 100
x_init = 50

X = firm.sim_inventory_path(x_init, sim_length)

fig, ax = plt.subplots()
bbox = (0., 1.02, 1., .102)
legend_args = {'ncol': 3,
               'bbox_to_anchor': bbox,
               'loc': 3,
               'mode': 'expand'}

ax.plot(X, label="inventory")
ax.plot(s * np.ones(sim_length), 'k--', label="$s$")
ax.plot(S * np.ones(sim_length), 'k-', label="$S$")
ax.set_ylim(0, S+10)
ax.set_xlabel("time")
ax.legend(**legend_args)

plt.show()
```

Now let's simulate multiple paths in order to build a more complete picture of the probabilities of different outcomes:

In [4]:
```
sim_length=200
fig, ax = plt.subplots()
```

```
ax.plot(s * np.ones(sim_length), 'k--', label="$s$")
ax.plot(S * np.ones(sim_length), 'k-', label="$S$")
ax.set_ylim(0, S+10)
ax.legend(**legend_args)

for i in range(400):
    X = firm.sim_inventory_path(x_init, sim_length)
    ax.plot(X, 'b', alpha=0.2, lw=0.5)

plt.show()
```

27.4 Marginal Distributions

Now let's look at the marginal distribution ψ_T of X_T for some fixed T.

We will do this by generating many draws of X_T given initial condition X_0.

With these draws of X_T we can build up a picture of its distribution ψ_T

Here's one visualization, with $T = 50$.

In [5]:
```
T = 50
M = 200  # Number of draws

ymin, ymax = 0, S + 10

fig, axes = plt.subplots(1, 2, figsize=(11, 6))

for ax in axes:
```

```
    ax.grid(alpha=0.4)

ax = axes[0]

ax.set_ylim(ymin, ymax)
ax.set_ylabel('$X_t$', fontsize=16)
ax.vlines((T,), -1.5, 1.5)

ax.set_xticks((T,))
ax.set_xticklabels((r'$T$',))

sample = np.empty(M)
for m in range(M):
    X = firm.sim_inventory_path(x_init, 2 * T)
    ax.plot(X, 'b-', lw=1, alpha=0.5)
    ax.plot((T,), (X[T+1],), 'ko', alpha=0.5)
    sample[m] = X[T+1]

axes[1].set_ylim(ymin, ymax)

axes[1].hist(sample,
             bins=16,
             density=True,
             orientation='horizontal',
             histtype='bar',
             alpha=0.5)

plt.show()
```

We can build up a clearer picture by drawing more samples

```
In [6]: T = 50
        M = 50_000

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

```
sample = np.empty(M)
for m in range(M):
    X = firm.sim_inventory_path(x_init, T+1)
    sample[m] = X[T]

ax.hist(sample,
         bins=36,
         density=True,
         histtype='bar',
         alpha=0.75)

plt.show()
```

Note that the distribution is bimodal

- Most firms have restocked twice but a few have restocked only once (see figure with paths above).
- Firms in the second category have lower inventory.

We can also approximate the distribution using a kernel density estimator.

Kernel density estimators can be thought of as smoothed histograms.

They are preferable to histograms when the distribution being estimated is likely to be smooth.

We will use a kernel density estimator from scikit-learn

In [7]:
```
from sklearn.neighbors import KernelDensity

def plot_kde(sample, ax, label=''):

    xmin, xmax = 0.9 * min(sample), 1.1 * max(sample)
    xgrid = np.linspace(xmin, xmax, 200)
    kde = KernelDensity(kernel='gaussian').fit(sample[:, None])
```

```
    log_dens = kde.score_samples(xgrid[:, None])

    ax.plot(xgrid, np.exp(log_dens), label=label)
```

```
In [8]: fig, ax = plt.subplots()
        plot_kde(sample, ax)
        plt.show()
```

The allocation of probability mass is similar to what was shown by the histogram just above.

27.5 Exercises

27.5.1 Exercise 1

This model is asymptotically stationary, with a unique stationary distribution.

(See the discussion of stationarity in our lecture on AR(1) processes for background — the fundamental concepts are the same.)

In particular, the sequence of marginal distributions $\{\psi_t\}$ is converging to a unique limiting distribution that does not depend on initial conditions.

Although we will not prove this here, we can investigate it using simulation.

Your task is to generate and plot the sequence $\{\psi_t\}$ at times $t = 10, 50, 250, 500, 750$ based on the discussion above.

(The kernel density estimator is probably the best way to present each distribution.)

You should see convergence, in the sense that differences between successive distributions are getting smaller.

Try different initial conditions to verify that, in the long run, the distribution is invariant across initial conditions.

27.5.2 Exercise 2

Using simulation, calculate the probability that firms that start with $X_0 = 70$ need to order twice or more in the first 50 periods.

You will need a large sample size to get an accurate reading.

27.6 Solutions

27.6.1 Exercise 1

Below is one possible solution:

The computations involve a lot of CPU cycles so we have tried to write the code efficiently.

This meant writing a specialized function rather than using the class above.

In [9]:
```
s, S, mu, sigma = firm.s, firm.S, firm.mu, firm.sigma

@njit(parallel=True)
def shift_firms_forward(current_inventory_levels, num_periods):

    num_firms = len(current_inventory_levels)
    new_inventory_levels = np.empty(num_firms)

    for f in prange(num_firms):
        x = current_inventory_levels[f]
        for t in range(num_periods):
            Z = np.random.randn()
            D = np.exp(mu + sigma * Z)
            if x <= s:
                x = max(S - D, 0)
            else:
                x = max(x - D, 0)
        new_inventory_levels[f] = x

    return new_inventory_levels
```

In [10]:
```
x_init = 50
num_firms = 50_000

sample_dates = 0, 10, 50, 250, 500, 750

first_diffs = np.diff(sample_dates)

fig, ax = plt.subplots()

X = np.ones(num_firms) * x_init

current_date = 0
for d in first_diffs:
    X = shift_firms_forward(X, d)
    current_date += d
    plot_kde(X, ax, label=f't = {current_date}')

ax.set_xlabel('inventory')
```

```
ax.set_ylabel('probability')
ax.legend()
plt.show()
```

Notice that by $t = 500$ or $t = 750$ the densities are barely changing.

We have reached a reasonable approximation of the stationary density.

You can convince yourself that initial conditions don't matter by testing a few of them.

For example, try rerunning the code above will all firms starting at $X_0 = 20$ or $X_0 = 80$.

27.6.2 Exercise 2

Here is one solution.

Again, the computations are relatively intensive so we have written a a specialized function rather than using the class above.

We will also use parallelization across firms.

```
In [11]: @njit(parallel=True)
         def compute_freq(sim_length=50, x_init=70, num_firms=1_000_000):

             firm_counter = 0  # Records number of firms that restock 2x or more
             for m in prange(num_firms):
                 x = x_init
                 restock_counter = 0  # Will record number of restocks for firm m

                 for t in range(sim_length):
                     Z = np.random.randn()
                     D = np.exp(mu + sigma * Z)
                     if x <= s:
```

```
                x = max(S - D, 0)
                restock_counter += 1
            else:
                x = max(x - D, 0)

        if restock_counter > 1:
            firm_counter += 1

    return firm_counter / num_firms
```

Note the time the routine takes to run, as well as the output.

```
In [12]: %%time

         freq = compute_freq()
         print(f"Frequency of at least two stock outs = {freq}")
```

```
    Frequency of at least two stock outs = 0.447685
CPU times: user 4.96 s, sys: 7.7 ms, total: 4.97 s
Wall time: 4 s
```

Try switching the `parallel` flag to `False` in the jitted function above.

Depending on your system, the difference can be substantial.

(On our desktop machine, the speed up is by a factor of 5.)

Chapter 28

Linear State Space Models

28.1 Contents

- Overview 28.2
- The Linear State Space Model 28.3
- Distributions and Moments 28.4
- Stationarity and Ergodicity 28.5
- Noisy Observations 28.6
- Prediction 28.7
- Code 28.8
- Exercises 28.9
- Solutions 28.10

> "We may regard the present state of the universe as the effect of its past and the cause of its future" – Marquis de Laplace

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

```
In [1]: !pip install --upgrade quantecon
```

28.2 Overview

This lecture introduces the **linear state space** dynamic system.

This model is a workhorse that carries a powerful theory of prediction.

Its many applications include:

- representing dynamics of higher-order linear systems
- predicting the position of a system j steps into the future
- predicting a geometric sum of future values of a variable like
  - non-financial income
  - dividends on a stock
  - the money supply
  - a government deficit or surplus, etc.
- key ingredient of useful models
  - Friedman's permanent income model of consumption smoothing.

  - Barro's model of smoothing total tax collections.
  - Rational expectations version of Cagan's model of hyperinflation.
  - Sargent and Wallace's "unpleasant monetarist arithmetic," etc.

Let's start with some imports:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        from quantecon import LinearStateSpace
        from scipy.stats import norm
        import random
```

28.3 The Linear State Space Model

The objects in play are:

- An $n \times 1$ vector x_t denoting the **state** at time $t = 0, 1, 2, \ldots$.
- An IID sequence of $m \times 1$ random vectors $w_t \sim N(0, I)$.
- A $k \times 1$ vector y_t of **observations** at time $t = 0, 1, 2, \ldots$.
- An $n \times n$ matrix A called the **transition matrix**.
- An $n \times m$ matrix C called the **volatility matrix**.
- A $k \times n$ matrix G sometimes called the **output matrix**.

Here is the linear state-space system

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t \\ x_0 &\sim N(\mu_0, \Sigma_0) \end{aligned} \tag{1}$$

28.3.1 Primitives

The primitives of the model are

1. the matrices A, C, G
2. shock distribution, which we have specialized to $N(0, I)$
3. the distribution of the initial condition x_0, which we have set to $N(\mu_0, \Sigma_0)$

Given A, C, G and draws of x_0 and $w_1, w_2, \ldots$, the model (1) pins down the values of the sequences $\{x_t\}$ and $\{y_t\}$.

Even without these draws, the primitives 1–3 pin down the *probability distributions* of $\{x_t\}$ and $\{y_t\}$.

Later we'll see how to compute these distributions and their moments.

Martingale Difference Shocks

We've made the common assumption that the shocks are independent standardized normal vectors.

But some of what we say will be valid under the assumption that $\{w_{t+1}\}$ is a **martingale difference sequence**.

A martingale difference sequence is a sequence that is zero mean when conditioned on past information.

In the present case, since $\{x_t\}$ is our state sequence, this means that it satisfies

$$\mathbb{E}[w_{t+1}|x_t, x_{t-1}, ...] = 0$$

This is a weaker condition than that $\{w_t\}$ is IID with $w_{t+1} \sim N(0, I)$.

28.3.2 Examples

By appropriate choice of the primitives, a variety of dynamics can be represented in terms of the linear state space model.

The following examples help to highlight this point.

They also illustrate the wise dictum *finding the state is an art*.

Second-order Difference Equation

Let $\{y_t\}$ be a deterministic sequence that satisfies

$$y_{t+1} = \phi_0 + \phi_1 y_t + \phi_2 y_{t-1} \quad \text{s.t.} \quad y_0, y_{-1} \text{ given} \tag{2}$$

To map (2) into our state space system (1), we set

$$x_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix} \qquad A = \begin{bmatrix} 1 & 0 & 0 \\ \phi_0 & \phi_1 & \phi_2 \\ 0 & 1 & 0 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

You can confirm that under these definitions, (1) and (2) agree.

The next figure shows the dynamics of this process when $\phi_0 = 1.1, \phi_1 = 0.8, \phi_2 = -0.8, y_0 =$

$y_{-1} = 1$.

Later you'll be asked to recreate this figure.

Univariate Autoregressive Processes

We can use (1) to represent the model

$$y_{t+1} = \phi_1 y_t + \phi_2 y_{t-1} + \phi_3 y_{t-2} + \phi_4 y_{t-3} + \sigma w_{t+1} \tag{3}$$

where $\{w_t\}$ is IID and standard normal.

To put this in the linear state space format we take $x_t = \begin{bmatrix} y_t & y_{t-1} & y_{t-2} & y_{t-3} \end{bmatrix}'$ and

$$A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

The matrix A has the form of the *companion matrix* to the vector $\begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \end{bmatrix}$.

The next figure shows the dynamics of this process when

$$\phi_1 = 0.5, \phi_2 = -0.2, \phi_3 = 0, \phi_4 = 0.5, \sigma = 0.2, y_0 = y_{-1} = y_{-2} = y_{-3} = 1$$

Vector Autoregressions

Now suppose that

- y_t is a $k \times 1$ vector
- ϕ_j is a $k \times k$ matrix and
- w_t is $k \times 1$

Then (3) is termed a *vector autoregression*.

To map this into (1), we set

$$x_t = \begin{bmatrix} y_t \\ y_{t-1} \\ y_{t-2} \\ y_{t-3} \end{bmatrix} \quad A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \end{bmatrix} \quad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad G = \begin{bmatrix} I & 0 & 0 & 0 \end{bmatrix}$$

where I is the $k \times k$ identity matrix and σ is a $k \times k$ matrix.

Seasonals

We can use (1) to represent

1. the *deterministic seasonal* $y_t = y_{t-4}$
2. the *indeterministic seasonal* $y_t = \phi_4 y_{t-4} + w_t$

In fact, both are special cases of (3).

With the deterministic seasonal, the transition matrix becomes

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

It is easy to check that $A^4 = I$, which implies that x_t is strictly periodic with period 4:Section **??**

$$x_{t+4} = x_t$$

Such an x_t process can be used to model deterministic seasonals in quarterly time series.

The *indeterministic* seasonal produces recurrent, but aperiodic, seasonal fluctuations.

Time Trends

The model $y_t = at + b$ is known as a *linear time trend.*

We can represent this model in the linear state space form by taking

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} a & b \end{bmatrix} \tag{4}$$

and starting at initial condition $x_0 = \begin{bmatrix} 0 & 1 \end{bmatrix}'$.

In fact, it's possible to use the state-space system to represent polynomial trends of any order.

For instance, we can represent the model $y_t = at^2 + bt + c$ in the linear state space form by taking

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} 2a & a+b & c \end{bmatrix}$$

and starting at initial condition $x_0 = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}'$.

It follows that

$$A^t = \begin{bmatrix} 1 & t & t(t-1)/2 \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix}$$

Then $x_t' = \begin{bmatrix} t(t-1)/2 & t & 1 \end{bmatrix}$. You can now confirm that $y_t = Gx_t$ has the correct form.

28.3.3 Moving Average Representations

A nonrecursive expression for x_t as a function of $x_0, w_1, w_2, \dots, w_t$ can be found by using (1) repeatedly to obtain

$$\begin{aligned} x_t &= Ax_{t-1} + Cw_t \\ &= A^2 x_{t-2} + ACw_{t-1} + Cw_t \\ &\quad \vdots \\ &= \sum_{j=0}^{t-1} A^j C w_{t-j} + A^t x_0 \end{aligned} \tag{5}$$

Representation (5) is a *moving average* representation.

It expresses $\{x_t\}$ as a linear function of

1. current and past values of the process $\{w_t\}$ and
2. the initial condition x_0

As an example of a moving average representation, let the model be

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

You will be able to show that $A^t = \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix}$ and $A^j C = \begin{bmatrix} 1 & 0 \end{bmatrix}'$.

Substituting into the moving average representation (5), we obtain

$$x_{1t} = \sum_{j=0}^{t-1} w_{t-j} + \begin{bmatrix} 1 & t \end{bmatrix} x_0$$

where x_{1t} is the first entry of x_t.

The first term on the right is a cumulated sum of martingale differences and is therefore a martingale.

The second term is a translated linear function of time.

For this reason, x_{1t} is called a *martingale with drift*.

28.4 Distributions and Moments

28.4.1 Unconditional Moments

Using (1), it's easy to obtain expressions for the (unconditional) means of x_t and y_t.

We'll explain what *unconditional* and *conditional* mean soon.

Letting $\mu_t := \mathbb{E}[x_t]$ and using linearity of expectations, we find that

$$\mu_{t+1} = A\mu_t \quad \text{with} \quad \mu_0 \text{ given} \tag{6}$$

Here μ_0 is a primitive given in (1).

The variance-covariance matrix of x_t is $\Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)']$.

Using $x_{t+1} - \mu_{t+1} = A(x_t - \mu_t) + Cw_{t+1}$, we can determine this matrix recursively via

$$\Sigma_{t+1} = A\Sigma_t A' + CC' \quad \text{with} \quad \Sigma_0 \text{ given} \tag{7}$$

As with μ_0, the matrix Σ_0 is a primitive given in (1).

As a matter of terminology, we will sometimes call

- μ_t the *unconditional mean* of x_t
- Σ_t the *unconditional variance-covariance matrix* of x_t

This is to distinguish μ_t and Σ_t from related objects that use conditioning information, to be defined below.

However, you should be aware that these "unconditional" moments do depend on the initial distribution $N(\mu_0, \Sigma_0)$.

Moments of the Observations

Using linearity of expectations again we have

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t] = G\mu_t \tag{8}$$

The variance-covariance matrix of y_t is easily shown to be

$$\mathrm{Var}[y_t] = \mathrm{Var}[Gx_t] = G\Sigma_t G' \tag{9}$$

28.4.2 Distributions

In general, knowing the mean and variance-covariance matrix of a random vector is not quite as good as knowing the full distribution.

However, there are some situations where these moments alone tell us all we need to know.

These are situations in which the mean vector and covariance matrix are **sufficient statistics** for the population distribution.

(Sufficient statistics form a list of objects that characterize a population distribution)

One such situation is when the vector in question is Gaussian (i.e., normally distributed).

This is the case here, given

1. our Gaussian assumptions on the primitives
2. the fact that normality is preserved under linear operations

In fact, it's well-known that

$$u \sim N(\bar{u}, S) \quad \text{and} \quad v = a + Bu \implies v \sim N(a + B\bar{u}, BSB') \tag{10}$$

In particular, given our Gaussian assumptions on the primitives and the linearity of (1) we can see immediately that both x_t and y_t are Gaussian for all $t \geq 0$ Section **??**.

Since x_t is Gaussian, to find the distribution, all we need to do is find its mean and variance-covariance matrix.

But in fact we've already done this, in (6) and (7).

Letting μ_t and Σ_t be as defined by these equations, we have

$$x_t \sim N(\mu_t, \Sigma_t) \tag{11}$$

By similar reasoning combined with (8) and (9),

$$y_t \sim N(G\mu_t, G\Sigma_t G') \tag{12}$$

28.4.3 Ensemble Interpretations

How should we interpret the distributions defined by (11)–(12)?

Intuitively, the probabilities in a distribution correspond to relative frequencies in a large population drawn from that distribution.

Let's apply this idea to our setting, focusing on the distribution of y_T for fixed T.

We can generate independent draws of y_T by repeatedly simulating the evolution of the system up to time T, using an independent set of shocks each time.

The next figure shows 20 simulations, producing 20 time series for $\{y_t\}$, and hence 20 draws of y_T.

The system in question is the univariate autoregressive model (3).

The values of y_T are represented by black dots in the left-hand figure

In the right-hand figure, these values are converted into a rotated histogram that shows relative frequencies from our sample of 20 y_T's.

(The parameters and source code for the figures can be found in file linear_models/paths_and_hist.py)

Here is another figure, this time with 100 observations

Let's now try with 500,000 observations, showing only the histogram (without rotation)

The black line is the population density of y_T calculated from (12).

The histogram and population distribution are close, as expected.

By looking at the figures and experimenting with parameters, you will gain a feel for how the population distribution depends on the model primitives listed above, as intermediated by the distribution's sufficient statistics.

Ensemble Means

In the preceding figure, we approximated the population distribution of y_T by

1. generating I sample paths (i.e., time series) where I is a large number
2. recording each observation y_T^i
3. histogramming this sample

Just as the histogram approximates the population distribution, the *ensemble* or *cross-sectional average*

$$\bar{y}_T := \frac{1}{I} \sum_{i=1}^{I} y_T^i$$

approximates the expectation $\mathbb{E}[y_T] = G\mu_T$ (as implied by the law of large numbers).

Here's a simulation comparing the ensemble averages and population means at time points $t = 0, \dots, 50$.

The parameters are the same as for the preceding figures, and the sample size is relatively

small ($I = 20$).

The ensemble mean for x_t is

$$\bar{x}_T := \frac{1}{I}\sum_{i=1}^{I} x_T^i \to \mu_T \qquad (I \to \infty)$$

The limit μ_T is a "long-run average".

(By *long-run average* we mean the average for an infinite ($I = \infty$) number of sample x_T's)

Another application of the law of large numbers assures us that

$$\frac{1}{I}\sum_{i=1}^{I}(x_T^i - \bar{x}_T)(x_T^i - \bar{x}_T)' \to \Sigma_T \qquad (I \to \infty)$$

28.4.4 Joint Distributions

In the preceding discussion, we looked at the distributions of x_t and y_t in isolation.

This gives us useful information but doesn't allow us to answer questions like

- what's the probability that $x_t \geq 0$ for all t?
- what's the probability that the process $\{y_t\}$ exceeds some value a before falling below b?
- etc., etc.

Such questions concern the *joint distributions* of these sequences.

To compute the joint distribution of $x_0, x_1, \ldots, x_T$, recall that joint and conditional densities are linked by the rule

$$p(x, y) = p(y \,|\, x)p(x) \qquad (\text{joint} \;=\; \text{conditional} \;\times\; \text{marginal})$$

From this rule we get $p(x_0, x_1) = p(x_1 \mid x_0)p(x_0)$.

The Markov property $p(x_t \mid x_{t-1}, \dots, x_0) = p(x_t \mid x_{t-1})$ and repeated applications of the preceding rule lead us to

$$p(x_0, x_1, \dots, x_T) = p(x_0) \prod_{t=0}^{T-1} p(x_{t+1} \mid x_t)$$

The marginal $p(x_0)$ is just the primitive $N(\mu_0, \Sigma_0)$.

In view of (1), the conditional densities are

$$p(x_{t+1} \mid x_t) = N(Ax_t, CC')$$

Autocovariance Functions

An important object related to the joint distribution is the *autocovariance function*

$$\Sigma_{t+j,t} := \mathbb{E}[(x_{t+j} - \mu_{t+j})(x_t - \mu_t)'] \tag{13}$$

Elementary calculations show that

$$\Sigma_{t+j,t} = A^j \Sigma_t \tag{14}$$

Notice that $\Sigma_{t+j,t}$ in general depends on both j, the gap between the two dates, and t, the earlier date.

28.5 Stationarity and Ergodicity

Stationarity and ergodicity are two properties that, when they hold, greatly aid analysis of linear state space models.

Let's start with the intuition.

28.5.1 Visualizing Stability

Let's look at some more time series from the same model that we analyzed above.

This picture shows cross-sectional distributions for y at times T, T', T''

Note how the time series "settle down" in the sense that the distributions at T' and T'' are relatively similar to each other — but unlike the distribution at T.

Apparently, the distributions of y_t converge to a fixed long-run distribution as $t \to \infty$.

When such a distribution exists it is called a *stationary distribution*.

28.5.2 Stationary Distributions

In our setting, a distribution ψ_∞ is said to be *stationary* for x_t if

$$x_t \sim \psi_\infty \quad \text{and} \quad x_{t+1} = Ax_t + Cw_{t+1} \quad \Longrightarrow \quad x_{t+1} \sim \psi_\infty$$

Since

1. in the present case, all distributions are Gaussian

2. a Gaussian distribution is pinned down by its mean and variance-covariance matrix

we can restate the definition as follows: ψ_∞ is stationary for x_t if

$$\psi_\infty = N(\mu_\infty, \Sigma_\infty)$$

where μ_∞ and Σ_∞ are fixed points of (6) and (7) respectively.

28.5.3 Covariance Stationary Processes

Let's see what happens to the preceding figure if we start x_0 at the stationary distribution.

Now the differences in the observed distributions at T, T' and T'' come entirely from random fluctuations due to the finite sample size.

By

- our choosing $x_0 \sim N(\mu_\infty, \Sigma_\infty)$
- the definitions of μ_∞ and Σ_∞ as fixed points of (6) and (7) respectively

we've ensured that

$$\mu_t = \mu_\infty \quad \text{and} \quad \Sigma_t = \Sigma_\infty \quad \text{for all } t$$

Moreover, in view of (14), the autocovariance function takes the form $\Sigma_{t+j,t} = A^j \Sigma_\infty$, which depends on j but not on t.

This motivates the following definition.

A process $\{x_t\}$ is said to be *covariance stationary* if

- both μ_t and Σ_t are constant in t
- $\Sigma_{t+j,t}$ depends on the time gap j but not on time t

In our setting, $\{x_t\}$ will be covariance stationary if μ_0, Σ_0, A, C assume values that imply that none of $\mu_t, \Sigma_t, \Sigma_{t+j,t}$ depends on t.

28.5.4 Conditions for Stationarity

The Globally Stable Case

The difference equation $\mu_{t+1} = A\mu_t$ is known to have *unique* fixed point $\mu_\infty = 0$ if all eigenvalues of A have moduli strictly less than unity.

That is, if `(np.absolute(np.linalg.eigvals(A)) < 1).all() == True`.

The difference equation (7) also has a unique fixed point in this case, and, moreover

$$\mu_t \to \mu_\infty = 0 \quad \text{and} \quad \Sigma_t \to \Sigma_\infty \quad \text{as} \quad t \to \infty$$

regardless of the initial conditions μ_0 and Σ_0.

This is the *globally stable case* — see these notes for more a theoretical treatment.

However, global stability is more than we need for stationary solutions, and often more than we want.

To illustrate, consider our second order difference equation example.

Here the state is $x_t = \begin{bmatrix} 1 & y_t & y_{t-1} \end{bmatrix}'$.

Because of the constant first component in the state vector, we will never have $\mu_t \to 0$.

How can we find stationary solutions that respect a constant state component?

Processes with a Constant State Component

To investigate such a process, suppose that A and C take the form

$$A = \begin{bmatrix} A_1 & a \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} C_1 \\ 0 \end{bmatrix}$$

where

- A_1 is an $(n-1) \times (n-1)$ matrix
- a is an $(n-1) \times 1$ column vector

Let $x_t = \begin{bmatrix} x_{1t}' & 1 \end{bmatrix}'$ where x_{1t} is $(n-1) \times 1$.

It follows that

$$x_{1,t+1} = A_1 x_{1t} + a + C_1 w_{t+1}$$

Let $\mu_{1t} = \mathbb{E}[x_{1t}]$ and take expectations on both sides of this expression to get

$$\mu_{1,t+1} = A_1 \mu_{1,t} + a \tag{15}$$

Assume now that the moduli of the eigenvalues of A_1 are all strictly less than one.

Then (15) has a unique stationary solution, namely,

$$\mu_{1\infty} = (I - A_1)^{-1} a$$

The stationary value of μ_t itself is then $\mu_\infty := \begin{bmatrix} \mu_{1\infty}' & 1 \end{bmatrix}'$.

The stationary values of Σ_t and $\Sigma_{t+j,t}$ satisfy

$$\begin{aligned} \Sigma_\infty &= A \Sigma_\infty A' + CC' \\ \Sigma_{t+j,t} &= A^j \Sigma_\infty \end{aligned} \tag{16}$$

Notice that here $\Sigma_{t+j,t}$ depends on the time gap j but not on calendar time t.

In conclusion, if

- $x_0 \sim N(\mu_\infty, \Sigma_\infty)$ and
- the moduli of the eigenvalues of A_1 are all strictly less than unity

then the $\{x_t\}$ process is covariance stationary, with constant state component.

> **Note**
>
> If the eigenvalues of A_1 are less than unity in modulus, then (a) starting from any initial value, the mean and variance-covariance matrix both converge to their stationary values; and (b) iterations on (7) converge to the fixed point of the *discrete Lyapunov equation* in the first line of (16).

28.5.5 Ergodicity

Let's suppose that we're working with a covariance stationary process.

In this case, we know that the ensemble mean will converge to μ_∞ as the sample size I approaches infinity.

Averages over Time

Ensemble averages across simulations are interesting theoretically, but in real life, we usually observe only a *single* realization $\{x_t, y_t\}_{t=0}^T$.

So now let's take a single realization and form the time-series averages

$$\bar{x} := \frac{1}{T}\sum_{t=1}^T x_t \quad \text{and} \quad \bar{y} := \frac{1}{T}\sum_{t=1}^T y_t$$

Do these time series averages converge to something interpretable in terms of our basic state-space representation?

The answer depends on something called *ergodicity*.

Ergodicity is the property that time series and ensemble averages coincide.

More formally, ergodicity implies that time series sample averages converge to their expectation under the stationary distribution.

In particular,

- $\frac{1}{T}\sum_{t=1}^T x_t \to \mu_\infty$
- $\frac{1}{T}\sum_{t=1}^T (x_t - \bar{x}_T)(x_t - \bar{x}_T)' \to \Sigma_\infty$
- $\frac{1}{T}\sum_{t=1}^T (x_{t+j} - \bar{x}_T)(x_t - \bar{x}_T)' \to A^j \Sigma_\infty$

In our linear Gaussian setting, any covariance stationary process is also ergodic.

28.6 Noisy Observations

In some settings, the observation equation $y_t = Gx_t$ is modified to include an error term.

Often this error term represents the idea that the true state can only be observed imperfectly.

To include an error term in the observation we introduce

- An IID sequence of $\ell \times 1$ random vectors $v_t \sim N(0, I)$.
- A $k \times \ell$ matrix H.

and extend the linear state-space system to

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + Hv_t \\ x_0 &\sim N(\mu_0, \Sigma_0) \end{aligned} \tag{17}$$

The sequence $\{v_t\}$ is assumed to be independent of $\{w_t\}$.

The process $\{x_t\}$ is not modified by noise in the observation equation and its moments, distributions and stability properties remain the same.

The unconditional moments of y_t from (8) and (9) now become

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t + Hv_t] = G\mu_t \tag{18}$$

The variance-covariance matrix of y_t is easily shown to be

$$\mathrm{Var}[y_t] = \mathrm{Var}[Gx_t + Hv_t] = G\Sigma_t G' + HH' \tag{19}$$

The distribution of y_t is therefore

$$y_t \sim N(G\mu_t, G\Sigma_t G' + HH')$$

28.7 Prediction

The theory of prediction for linear state space systems is elegant and simple.

28.7.1 Forecasting Formulas – Conditional Means

The natural way to predict variables is to use conditional distributions.

For example, the optimal forecast of x_{t+1} given information known at time t is

$$\mathbb{E}_t[x_{t+1}] := \mathbb{E}[x_{t+1} \mid x_t, x_{t-1}, \dots, x_0] = Ax_t$$

The right-hand side follows from $x_{t+1} = Ax_t + Cw_{t+1}$ and the fact that w_{t+1} is zero mean and independent of $x_t, x_{t-1}, \dots, x_0$.

That $\mathbb{E}_t[x_{t+1}] = \mathbb{E}[x_{t+1} \mid x_t]$ is an implication of $\{x_t\}$ having the *Markov property*.

The one-step-ahead forecast error is

$$x_{t+1} - \mathbb{E}_t[x_{t+1}] = Cw_{t+1}$$

The covariance matrix of the forecast error is

$$\mathbb{E}[(x_{t+1} - \mathbb{E}_t[x_{t+1}])(x_{t+1} - \mathbb{E}_t[x_{t+1}])'] = CC'$$

More generally, we'd like to compute the j-step ahead forecasts $\mathbb{E}_t[x_{t+j}]$ and $\mathbb{E}_t[y_{t+j}]$.

With a bit of algebra, we obtain

$$x_{t+j} = A^j x_t + A^{j-1} C w_{t+1} + A^{j-2} C w_{t+2} + \cdots + A^0 C w_{t+j}$$

In view of the IID property, current and past state values provide no information about future values of the shock.

Hence $\mathbb{E}_t[w_{t+k}] = \mathbb{E}[w_{t+k}] = 0$.

It now follows from linearity of expectations that the j-step ahead forecast of x is

$$\mathbb{E}_t[x_{t+j}] = A^j x_t$$

The j-step ahead forecast of y is therefore

$$\mathbb{E}_t[y_{t+j}] = \mathbb{E}_t[G x_{t+j} + H v_{t+j}] = G A^j x_t$$

28.7.2 Covariance of Prediction Errors

It is useful to obtain the covariance matrix of the vector of j-step-ahead prediction errors

$$x_{t+j} - \mathbb{E}_t[x_{t+j}] = \sum_{s=0}^{j-1} A^s C w_{t-s+j} \tag{20}$$

Evidently,

$$V_j := \mathbb{E}_t[(x_{t+j} - \mathbb{E}_t[x_{t+j}])(x_{t+j} - \mathbb{E}_t[x_{t+j}])'] = \sum_{k=0}^{j-1} A^k C C' A^{k'} \tag{21}$$

V_j defined in (21) can be calculated recursively via $V_1 = CC'$ and

$$V_j = CC' + A V_{j-1} A', \quad j \geq 2 \tag{22}$$

V_j is the *conditional covariance matrix* of the errors in forecasting x_{t+j}, conditioned on time t information x_t.

Under particular conditions, V_j converges to

$$V_\infty = CC' + A V_\infty A' \tag{23}$$

Equation (23) is an example of a *discrete Lyapunov* equation in the covariance matrix V_∞.

A sufficient condition for V_j to converge is that the eigenvalues of A be strictly less than one in modulus.

Weaker sufficient conditions for convergence associate eigenvalues equaling or exceeding one in modulus with elements of C that equal 0.

28.7.3 Forecasts of Geometric Sums

In several contexts, we want to compute forecasts of geometric sums of future random variables governed by the linear state-space system (1).

We want the following objects

- Forecast of a geometric sum of future x's, or $\mathbb{E}_t \left[\sum_{j=0}^{\infty} \beta^j x_{t+j}\right]$.
- Forecast of a geometric sum of future y's, or $\mathbb{E}_t \left[\sum_{j=0}^{\infty} \beta^j y_{t+j}\right]$.

These objects are important components of some famous and interesting dynamic models.

For example,

- if $\{y_t\}$ is a stream of dividends, then $\mathbb{E}\left[\sum_{j=0}^{\infty} \beta^j y_{t+j} | x_t\right]$ is a model of a stock price
- if $\{y_t\}$ is the money supply, then $\mathbb{E}\left[\sum_{j=0}^{\infty} \beta^j y_{t+j} | x_t\right]$ is a model of the price level

Formulas

Fortunately, it is easy to use a little matrix algebra to compute these objects.

Suppose that every eigenvalue of A has modulus strictly less than $\frac{1}{\beta}$.

It then follows that $I + \beta A + \beta^2 A^2 + \cdots = [I - \beta A]^{-1}$.

This leads to our formulas:

- Forecast of a geometric sum of future x's

$$\mathbb{E}_t \left[\sum_{j=0}^{\infty} \beta^j x_{t+j}\right] = [I + \beta A + \beta^2 A^2 + \cdots]x_t = [I - \beta A]^{-1} x_t$$

- Forecast of a geometric sum of future y's

$$\mathbb{E}_t \left[\sum_{j=0}^{\infty} \beta^j y_{t+j}\right] = G[I + \beta A + \beta^2 A^2 + \cdots]x_t = G[I - \beta A]^{-1} x_t$$

28.8 Code

Our preceding simulations and calculations are based on code in the file lss.py from the QuantEcon.py package.

The code implements a class for handling linear state space models (simulations, calculating moments, etc.).

One Python construct you might not be familiar with is the use of a generator function in the method `moment_sequence()`.

Go back and read the relevant documentation if you've forgotten how generator functions work.

Examples of usage are given in the solutions to the exercises.

28.9 Exercises

28.9.1 Exercise 1

Replicate this figure using the `LinearStateSpace` class from `lss.py`.

28.9.2 Exercise 2

Replicate this figure modulo randomness using the same class.

28.9.3 Exercise 3

Replicate this figure modulo randomness using the same class.

The state space model and parameters are the same as for the preceding exercise.

28.9.4 Exercise 4

Replicate this figure modulo randomness using the same class.

The state space model and parameters are the same as for the preceding exercise, except that the initial condition is the stationary distribution.

Hint: You can use the `stationary_distributions` method to get the initial conditions.

The number of sample paths is 80, and the time horizon in the figure is 100.

Producing the vertical bars and dots is optional, but if you wish to try, the bars are at dates 10, 50 and 75.

28.10 Solutions

28.10.1 Exercise 1

In [3]:
```
ϕ_0, ϕ_1, ϕ_2 = 1.1, 0.8, -0.8

A = [[1,     0,     0  ],
     [ϕ_0,   ϕ_1,   ϕ_2],
     [0,     1,     0  ]]
C = np.zeros((3, 1))
G = [0, 1, 0]

ar = LinearStateSpace(A, C, G, mu_0=np.ones(3))
x, y = ar.simulate(ts_length=50)

fig, ax = plt.subplots(figsize=(10, 6))
y = y.flatten()
ax.plot(y, 'b-', lw=2, alpha=0.7)
ax.grid()
```

```
ax.set_xlabel('time')
ax.set_ylabel('$y_t$', fontsize=16)
plt.show()
```

28.10.2 Exercise 2

```
In [4]: ϕ_1, ϕ_2, ϕ_3, ϕ_4 = 0.5, -0.2, 0, 0.5
        σ = 0.2

        A = [[ϕ_1,   ϕ_2,   ϕ_3,   ϕ_4],
             [1,     0,     0,     0  ],
             [0,     1,     0,     0  ],
             [0,     0,     1,     0  ]]
        C = [[σ],
             [0],
             [0],
             [0]]
        G = [1, 0, 0, 0]

        ar = LinearStateSpace(A, C, G, mu_0=np.ones(4))
        x, y = ar.simulate(ts_length=200)

        fig, ax = plt.subplots(figsize=(10, 6))
        y = y.flatten()
        ax.plot(y, 'b-', lw=2, alpha=0.7)
        ax.grid()
        ax.set_xlabel('time')
        ax.set_ylabel('$y_t$', fontsize=16)
        plt.show()
```

28.10.3 Exercise 3

```
In [5]: ϕ_1, ϕ_2, ϕ_3, ϕ_4 = 0.5, -0.2, 0, 0.5
        σ = 0.1

        A = [[ϕ_1,   ϕ_2,   ϕ_3,   ϕ_4],
             [1,     0,     0,     0  ],
             [0,     1,     0,     0  ],
             [0,     0,     1,     0  ]]
        C = [[σ],
             [0],
             [0],
             [0]]
        G = [1, 0, 0, 0]

        I = 20
        T = 50
        ar = LinearStateSpace(A, C, G, mu_0=np.ones(4))
        ymin, ymax = -0.5, 1.15

        fig, ax = plt.subplots(figsize=(8, 5))

        ax.set_ylim(ymin, ymax)
        ax.set_xlabel('time', fontsize=16)
        ax.set_ylabel('$y_t$', fontsize=16)

        ensemble_mean = np.zeros(T)
        for i in range(I):
            x, y = ar.simulate(ts_length=T)
            y = y.flatten()
            ax.plot(y, 'c-', lw=0.8, alpha=0.5)
            ensemble_mean = ensemble_mean + y
```

```
ensemble_mean = ensemble_mean / I
ax.plot(ensemble_mean, color='b', lw=2, alpha=0.8, label='$\\bar y_t$')

m = ar.moment_sequence()
population_means = []
for t in range(T):
    μ_x, μ_y, Σ_x, Σ_y = next(m)
    population_means.append(float(μ_y))
ax.plot(population_means, color='g', lw=2, alpha=0.8, label='$G\mu_t$')
ax.legend(ncol=2)
plt.show()
```

28.10.4 Exercise 4

```
In [6]: ϕ_1, ϕ_2, ϕ_3, ϕ_4 = 0.5, -0.2, 0, 0.5
        σ = 0.1

        A = [[ϕ_1,   ϕ_2,   ϕ_3,   ϕ_4],
             [1,     0,     0,     0  ],
             [0,     1,     0,     0  ],
             [0,     0,     1,     0  ]]
        C = [[σ],
             [0],
             [0],
             [0]]
        G = [1, 0, 0, 0]

        T0 = 10
        T1 = 50
        T2 = 75
        T4 = 100
```

```
ar = LinearStateSpace(A, C, G, mu_0=np.ones(4), Sigma_0=Σ_x)
ymin, ymax = -0.6, 0.6

fig, ax = plt.subplots(figsize=(8, 5))

ax.grid(alpha=0.4)
ax.set_ylim(ymin, ymax)
ax.set_ylabel('$y_t$', fontsize=16)
ax.vlines((T0, T1, T2), -1.5, 1.5)

ax.set_xticks((T0, T1, T2))
ax.set_xticklabels(("$T$", "$T'$", "$T''$"), fontsize=14)

μ_x, μ_y, Σ_x, Σ_y = ar.stationary_distributions()
ar.mu_0 = μ_x
ar.Sigma_0 = Σ_x

for i in range(80):
    rcolor = random.choice(('c', 'g', 'b'))
    x, y = ar.simulate(ts_length=T4)
    y = y.flatten()
    ax.plot(y, color=rcolor, lw=0.8, alpha=0.5)
    ax.plot((T0, T1, T2), (y[T0], y[T1], y[T2],), 'ko', alpha=0.5)
plt.show()
```

Footnotes

[**1**] The eigenvalues of A are $(1, -1, i, -i)$.

[**2**] The correct way to argue this is by induction. Suppose that x_t is Gaussian. Then (1) and (10) imply that x_{t+1} is Gaussian. Since x_0 is assumed to be Gaussian, it follows that every x_t is Gaussian. Evidently, this implies that each y_t is Gaussian.

Chapter 29

Application: The Samuelson Multiplier-Accelerator

29.1 Contents

- Overview 29.2
- Details 29.3
- Implementation 29.4
- Stochastic Shocks 29.5
- Government Spending 29.6
- Wrapping Everything Into a Class 29.7
- Using the LinearStateSpace Class 29.8
- Pure Multiplier Model 29.9
- Summary 29.10

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

```
In [1]: !pip install --upgrade quantecon
```

This lecture was created jointly with Natasha Watkins.

29.2 Overview

This lecture creates non-stochastic and stochastic versions of Paul Samuelson's celebrated multiplier accelerator model [139].

In doing so, we extend the example of the Solow model class in our second OOP lecture.

Our objectives are to

- provide a more detailed example of OOP and classes
- review a famous model
- review linear difference equations, both deterministic and stochastic

Let's start with some standard imports:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

We'll also use the following for various tasks described below:

```
In [3]: from quantecon import LinearStateSpace
        import cmath
        import math
        import sympy
        from sympy import Symbol, init_printing
        from cmath import sqrt
```

29.2.1 Samuelson's Model

Samuelson used a *second-order linear difference equation* to represent a model of national output based on three components:

- a *national output identity* asserting that national outcome is the sum of consumption plus investment plus government purchases.
- a Keynesian *consumption function* asserting that consumption at time t is equal to a constant times national output at time $t-1$.
- an investment *accelerator* asserting that investment at time t equals a constant called the *accelerator coefficient* times the difference in output between period $t-1$ and $t-2$.
- the idea that consumption plus investment plus government purchases constitute *aggregate demand,* which automatically calls forth an equal amount of *aggregate supply.*

(To read about linear difference equations see here or chapter IX of [142])

Samuelson used the model to analyze how particular values of the marginal propensity to consume and the accelerator coefficient might give rise to transient *business cycles* in national output.

Possible dynamic properties include

- smooth convergence to a constant level of output
- damped business cycles that eventually converge to a constant level of output
- persistent business cycles that neither dampen nor explode

Later we present an extension that adds a random shock to the right side of the national income identity representing random fluctuations in aggregate demand.

This modification makes national output become governed by a second-order *stochastic linear difference equation* that, with appropriate parameter values, gives rise to recurrent irregular business cycles.

(To read about stochastic linear difference equations see chapter XI of [142])

29.3 Details

Let's assume that

- $\{G_t\}$ is a sequence of levels of government expenditures – we'll start by setting $G_t = G$ for all t.
- $\{C_t\}$ is a sequence of levels of aggregate consumption expenditures, a key endogenous variable in the model.
- $\{I_t\}$ is a sequence of rates of investment, another key endogenous variable.
- $\{Y_t\}$ is a sequence of levels of national income, yet another endogenous variable.

- a is the marginal propensity to consume in the Keynesian consumption function $C_t = aY_{t-1} + \gamma$.
- b is the "accelerator coefficient" in the "investment accelerator" $I_t = b(Y_t - 1 - Y_t - 2)$.
- $\{\epsilon_t\}$ is an IID sequence standard normal random variables.
- $\sigma \geq 0$ is a "volatility" parameter — setting $\sigma = 0$ recovers the non-stochastic case that we'll start with.

The model combines the consumption function

$$C_t = aY_{t-1} + \gamma \tag{1}$$

with the investment accelerator

$$I_t = b(Y_{t-1} - Y_{t-2}) \tag{2}$$

and the national income identity

$$Y_t = C_t + I_t + G_t \tag{3}$$

- The parameter a is peoples' *marginal propensity to consume* out of income - equation (1) asserts that people consume a fraction of math:a in (0,1) of each additional dollar of income.
- The parameter $b > 0$ is the investment accelerator coefficient - equation (2) asserts that people invest in physical capital when income is increasing and disinvest when it is decreasing.

Equations (1), (2), and (3) imply the following second-order linear difference equation for national income:

$$Y_t = (a + b)Y_{t-1} - bY_{t-2} + (\gamma + G_t)$$

or

$$Y_t = \rho_1 Y_{t-1} + \rho_2 Y_{t-2} + (\gamma + G_t) \tag{4}$$

where $\rho_1 = (a + b)$ and $\rho_2 = -b$.

To complete the model, we require two **initial conditions**.

If the model is to generate time series for $t = 0, \dots, T$, we require initial values

$$Y_{-1} = \bar{Y}_{-1}, \quad Y_{-2} = \bar{Y}_{-2}$$

We'll ordinarily set the parameters (a, b) so that starting from an arbitrary pair of initial conditions $(\bar{Y}_{-1}, \bar{Y}_{-2})$, national income Y_t converges to a constant value as t becomes large.

We are interested in studying

- the transient fluctuations in Y_t as it converges to its **steady state** level
- the **rate** at which it converges to a steady state level

The deterministic version of the model described so far — meaning that no random shocks hit aggregate demand — has only transient fluctuations.

We can convert the model to one that has persistent irregular fluctuations by adding a random shock to aggregate demand.

29.3.1 Stochastic Version of the Model

We create a **random** or **stochastic** version of the model by adding a random process of **shocks** or **disturbances** $\{\sigma\epsilon_t\}$ to the right side of equation (4), leading to the **second-order scalar linear stochastic difference equation**:

$$Y_t = G_t + a(1-b)Y_{t-1} - abY_{t-2} + \sigma\epsilon_t \tag{5}$$

29.3.2 Mathematical Analysis of the Model

To get started, let's set $G_t \equiv 0$, $\sigma = 0$, and $\gamma = 0$.

Then we can write equation (5) as

$$Y_t = \rho_1 Y_{t-1} + \rho_2 Y_{t-2}$$

or

$$Y_{t+2} - \rho_1 Y_{t+1} - \rho_2 Y_t = 0 \tag{6}$$

To discover the properties of the solution of (6), it is useful first to form the **characteristic polynomial** for (6):

$$z^2 - \rho_1 z - \rho_2 \tag{7}$$

where z is possibly a complex number.

We want to find the two **zeros** (a.k.a. **roots**) – namely λ_1, λ_2 – of the characteristic polynomial.

These are two special values of z, say $z = \lambda_1$ and $z = \lambda_2$, such that if we set z equal to one of these values in expression (7), the characteristic polynomial (7) equals zero:

$$z^2 - \rho_1 z - \rho_2 = (z - \lambda_1)(z - \lambda_2) = 0 \tag{8}$$

Equation (8) is said to **factor** the characteristic polynomial.

When the roots are complex, they will occur as a complex conjugate pair.

When the roots are complex, it is convenient to represent them in the polar form

$$\lambda_1 = re^{i\omega}, \ \lambda_2 = re^{-i\omega}$$

where r is the *amplitude* of the complex number and ω is its *angle* or *phase*.

These can also be represented as

$$\lambda_1 = r(cos(\omega) + i\sin(\omega))$$

$$\lambda_2 = r(cos(\omega) - i\sin(\omega))$$

(To read about the polar form, see here)

Given **initial conditions** Y_{-1}, Y_{-2}, we want to generate a **solution** of the difference equation (6).

It can be represented as

$$Y_t = \lambda_1^t c_1 + \lambda_2^t c_2$$

where c_1 and c_2 are constants that depend on the two initial conditions and on ρ_1, ρ_2.

When the roots are complex, it is useful to pursue the following calculations.

Notice that

$$\begin{aligned} Y_t &= c_1(re^{i\omega})^t + c_2(re^{-i\omega})^t \\ &= c_1 r^t e^{i\omega t} + c_2 r^t e^{-i\omega t} \\ &= c_1 r^t[\cos(\omega t) + i\sin(\omega t)] + c_2 r^t[\cos(\omega t) - i\sin(\omega t)] \\ &= (c_1 + c_2) r^t \cos(\omega t) + i(c_1 - c_2) r^t \sin(\omega t) \end{aligned}$$

The only way that Y_t can be a real number for each t is if $c_1 + c_2$ is a real number and $c_1 - c_2$ is an imaginary number.

This happens only when c_1 and c_2 are complex conjugates, in which case they can be written in the polar forms

$$c_1 = ve^{i\theta}, \quad c_2 = ve^{-i\theta}$$

So we can write

$$\begin{aligned} Y_t &= ve^{i\theta} r^t e^{i\omega t} + ve^{-i\theta} r^t e^{-i\omega t} \\ &= vr^t[e^{i(\omega t+\theta)} + e^{-i(\omega t+\theta)}] \\ &= 2vr^t \cos(\omega t + \theta) \end{aligned}$$

where v and θ are constants that must be chosen to satisfy initial conditions for Y_{-1}, Y_{-2}.

This formula shows that when the roots are complex, Y_t displays oscillations with **period** $\check{p} = \frac{2\pi}{\omega}$ and **damping factor** r.

We say that $\check{p}$ is the **period** because in that amount of time the cosine wave $\cos(\omega t + \theta)$ goes through exactly one complete cycles.

(Draw a cosine function to convince yourself of this please)

Remark: Following [139], we want to choose the parameters a, b of the model so that the absolute values (of the possibly complex) roots λ_1, λ_2 of the characteristic polynomial are both strictly less than one:

$$|\lambda_j| < 1 \qquad \text{for } j = 1, 2$$

Remark: When both roots λ_1, λ_2 of the characteristic polynomial have absolute values strictly less than one, the absolute value of the larger one governs the rate of convergence to the steady state of the non stochastic version of the model.

29.3.3 Things This Lecture Does

We write a function to generate simulations of a $\{Y_t\}$ sequence as a function of time.

The function requires that we put in initial conditions for Y_{-1}, Y_{-2}.

The function checks that a, b are set so that λ_1, λ_2 are less than unity in absolute value (also called "modulus").

The function also tells us whether the roots are complex, and, if they are complex, returns both their real and complex parts.

If the roots are both real, the function returns their values.

We use our function written to simulate paths that are stochastic (when $\sigma > 0$).

We have written the function in a way that allows us to input $\{G_t\}$ paths of a few simple forms, e.g.,

- one time jumps in G at some time
- a permanent jump in G that occurs at some time

We proceed to use the Samuelson multiplier-accelerator model as a laboratory to make a simple OOP example.

The "state" that determines next period's Y_{t+1} is now not just the current value Y_t but also the once lagged value Y_{t-1}.

This involves a little more bookkeeping than is required in the Solow model class definition.

We use the Samuelson multiplier-accelerator model as a vehicle for teaching how we can gradually add more features to the class.

We want to have a method in the class that automatically generates a simulation, either non-stochastic ($\sigma = 0$) or stochastic ($\sigma > 0$).

We also show how to map the Samuelson model into a simple instance of the `LinearStateSpace` class described here.

We can use a `LinearStateSpace` instance to do various things that we did above with our homemade function and class.

Among other things, we show by example that the eigenvalues of the matrix A that we use to form the instance of the `LinearStateSpace` class for the Samuelson model equal the roots of the characteristic polynomial (7) for the Samuelson multiplier accelerator model.

Here is the formula for the matrix A in the linear state space system in the case that government expenditures are a constant G:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ \gamma + G & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}$$

29.4 Implementation

We'll start by drawing an informative graph from page 189 of [142]

```
In [4]: def param_plot():

            """This function creates the graph on page 189 of
            Sargent Macroeconomic Theory, second edition, 1987.
            """

            fig, ax = plt.subplots(figsize=(10, 6))
            ax.set_aspect('equal')

            # Set axis
            xmin, ymin = -3, -2
            xmax, ymax = -xmin, -ymin
            plt.axis([xmin, xmax, ymin, ymax])

            # Set axis labels
            ax.set(xticks=[], yticks=[])
            ax.set_xlabel(r'$\rho_2$', fontsize=16)
            ax.xaxis.set_label_position('top')
            ax.set_ylabel(r'$\rho_1$', rotation=0, fontsize=16)
            ax.yaxis.set_label_position('right')

            # Draw (t1, t2) points
            ρ1 = np.linspace(-2, 2, 100)
            ax.plot(ρ1, -abs(ρ1) + 1, c='black')
            ax.plot(ρ1, np.ones_like(ρ1) * -1, c='black')
            ax.plot(ρ1, -(ρ1**2 / 4), c='black')

            # Turn normal axes off
            for spine in ['left', 'bottom', 'top', 'right']:
                ax.spines[spine].set_visible(False)

            # Add arrows to represent axes
            axes_arrows = {'arrowstyle': '<|-|>', 'lw': 1.3}
            ax.annotate('', xy=(xmin, 0), xytext=(xmax, 0), arrowprops=axes_arrows)
            ax.annotate('', xy=(0, ymin), xytext=(0, ymax), arrowprops=axes_arrows)

            # Annotate the plot with equations
            plot_arrowsl = {'arrowstyle': '-|>', 'connectionstyle': "arc3, rad=-0.
 ↪2"}
            plot_arrowsr = {'arrowstyle': '-|>', 'connectionstyle': "arc3, rad=0.2"}
            ax.annotate(r'$\rho_1 + \rho_2 < 1$', xy=(0.5, 0.3), xytext=(0.8, 0.6),
                        arrowprops=plot_arrowsr, fontsize='12')
            ax.annotate(r'$\rho_1 + \rho_2 = 1$', xy=(0.38, 0.6), xytext=(0.6, 0.8),
                        arrowprops=plot_arrowsr, fontsize='12')
            ax.annotate(r'$\rho_2 < 1 + \rho_1$', xy=(-0.5, 0.3), xytext=(-1.3, 0.
 ↪6),
                        arrowprops=plot_arrowsl, fontsize='12')
            ax.annotate(r'$\rho_2 = 1 + \rho_1$', xy=(-0.38, 0.6), xytext=(-1, 0.8),
                        arrowprops=plot_arrowsl, fontsize='12')
            ax.annotate(r'$\rho_2 = -1$', xy=(1.5, -1), xytext=(1.8, -1.3),
                        arrowprops=plot_arrowsl, fontsize='12')
            ax.annotate(r'${\rho_1}^2 + 4\rho_2 = 0$', xy=(1.15, -0.35),
                        xytext=(1.5, -0.3), arrowprops=plot_arrowsr, fontsize='12')
```

```
    ax.annotate(r'${\rho_1}^2 + 4\rho_2 < 0$', xy=(1.4, -0.7),
                xytext=(1.8, -0.6), arrowprops=plot_arrowsr, fontsize='12')

    # Label categories of solutions
    ax.text(1.5, 1, 'Explosive\n growth', ha='center', fontsize=16)
    ax.text(-1.5, 1, 'Explosive\n oscillations', ha='center', fontsize=16)
    ax.text(0.05, -1.5, 'Explosive oscillations', ha='center', fontsize=16)
    ax.text(0.09, -0.5, 'Damped oscillations', ha='center', fontsize=16)

    # Add small marker to y-axis
    ax.axhline(y=1.005, xmin=0.495, xmax=0.505, c='black')
    ax.text(-0.12, -1.12, '-1', fontsize=10)
    ax.text(-0.12, 0.98, '1', fontsize=10)

    return fig

param_plot()
plt.show()
```

The graph portrays regions in which the (λ_1, λ_2) root pairs implied by the $(\rho_1 = (a+b), \rho_2 = -b)$ difference equation parameter pairs in the Samuelson model are such that:

- (λ_1, λ_2) are complex with modulus less than 1 - in this case, the $\{Y_t\}$ sequence displays damped oscillations.
- (λ_1, λ_2) are both real, but one is strictly greater than 1 - this leads to explosive growth.
- (λ_1, λ_2) are both real, but one is strictly less than -1 - this leads to explosive oscillations.
- (λ_1, λ_2) are both real and both are less than 1 in absolute value - in this case, there is smooth convergence to the steady state without damped cycles.

Later we'll present the graph with a red mark showing the particular point implied by the

setting of (a, b).

29.4.1 Function to Describe Implications of Characteristic Polynomial

```
In [5]: def categorize_solution(ρ1, ρ2):

            """This function takes values of ϱ1 and ϱ2 and uses them
            to classify the type of solution
            """

            discriminant = ρ1 ** 2 + 4 * ρ2
            if ρ2 > 1 + ρ1 or ρ2 < -1:
                print('Explosive oscillations')
            elif ρ1 + ρ2 > 1:
                print('Explosive growth')
            elif discriminant < 0:
                print('Roots are complex with modulus less than one; \
        therefore damped oscillations')
            else:
                print('Roots are real and absolute values are less than one; \
        therefore get smooth convergence to a steady state')
```

```
In [6]: ### Test the categorize_solution function

        categorize_solution(1.3, -.4)
```

```
   Roots are real and absolute values are less than one; therefore get smooth
 ↪convergence
to a steady state
```

29.4.2 Function for Plotting Paths

A useful function for our work below is

```
In [7]: def plot_y(function=None):

            """Function plots path of Y_t"""

            plt.subplots(figsize=(10, 6))
            plt.plot(function)
            plt.xlabel('Time $t$')
            plt.ylabel('$Y_t$', rotation=0)
            plt.grid()
            plt.show()
```

29.4.3 Manual or "by hand" Root Calculations

The following function calculates roots of the characteristic polynomial using high school algebra.

(We'll calculate the roots in other ways later)

The function also plots a Y_t starting from initial conditions that we set

In [8]:
```
# This is a 'manual' method

def y_nonstochastic(y_0=100, y_1=80, α=.92, β=.5, γ=10, n=80):

    """Takes values of parameters and computes the roots of characteristic
    polynomial. It tells whether they are real or complex and whether they
    are less than unity in absolute value.It also computes a simulation of
    length n starting from the two given initial conditions for national
    income
    """

    roots = []

    ρ1 = α + β
    ρ2 = -β

    print(f'ρ_1 is {ρ1}')
    print(f'ρ_2 is {ρ2}')

    discriminant = ρ1 ** 2 + 4 * ρ2

    if discriminant == 0:
        roots.append(-ρ1 / 2)
        print('Single real root: ')
        print(''.join(str(roots)))
    elif discriminant > 0:
        roots.append((-ρ1 + sqrt(discriminant).real) / 2)
        roots.append((-ρ1 - sqrt(discriminant).real) / 2)
        print('Two real roots: ')
        print(''.join(str(roots)))
    else:
        roots.append((-ρ1 + sqrt(discriminant)) / 2)
        roots.append((-ρ1 - sqrt(discriminant)) / 2)
        print('Two complex roots: ')
        print(''.join(str(roots)))

    if all(abs(root) < 1 for root in roots):
        print('Absolute values of roots are less than one')
    else:
        print('Absolute values of roots are not less than one')

    def transition(x, t): return ρ1 * x[t - 1] + ρ2 * x[t - 2] + γ

    y_t = [y_0, y_1]

    for t in range(2, n):
        y_t.append(transition(y_t, t))

    return y_t

plot_y(y_nonstochastic())
```

```
ρ_1 is 1.42
ρ_2 is -0.5
Two real roots:
[-0.6459687576256715, -0.7740312423743284]
Absolute values of roots are less than one
```

29.4.4 Reverse-Engineering Parameters to Generate Damped Cycles

The next cell writes code that takes as inputs the modulus r and phase ϕ of a conjugate pair of complex numbers in polar form

$$\lambda_1 = r\exp(i\phi), \quad \lambda_2 = r\exp(-i\phi)$$

- The code assumes that these two complex numbers are the roots of the characteristic polynomial
- It then reverse-engineers (a, b) and (ρ_1, ρ_2), pairs that would generate those roots

In [9]:
```
### code to reverse-engineer a cycle
### y_t = r^t (c_1 cos(⊠ t) + c2 sin(⊠ t))
###

def f(r, ϕ):
    """
    Takes modulus r and angle ⊠ of complex number r exp(j ⊠)
    and creates ϱ1 and ϱ2 of characteristic polynomial for which
    r exp(j ⊠) and r exp(- j ⊠) are complex roots.

    Returns the multiplier coefficient a and the accelerator coefficient b
    that verifies those roots.
    """
    g1 = cmath.rect(r, ϕ)  # Generate two complex roots
    g2 = cmath.rect(r, -ϕ)
    ρ1 = g1 + g2           # Implied ϱ1, ϱ2
    ρ2 = -g1 * g2
    b = -ρ2                # Reverse-engineer a and b that validate these
    a = ρ1 - b
    return ρ1, ρ2, a, b
```

```
## Now let's use the function in an example
## Here are the example parameters

r = .95
period = 10                # Length of cycle in units of time
ϕ = 2 * math.pi/period

## Apply the function

ρ1, ρ2, a, b = f(r, ϕ)

print(f"a, b = {a}, {b}")
print(f"ρ1, ρ2 = {ρ1}, {ρ2}")
```

```
    a, b = (0.6346322893124001+0j), (0.9024999999999999-0j)
ρ1, ρ2 = (1.5371322893124+0j), (-0.9024999999999999+0j)
```

In [10]:
```
## Print the real components of ρ1 and ρ2

ρ1 = ρ1.real
ρ2 = ρ2.real

ρ1, ρ2
```

Out[10]:
```
(1.5371322893124, -0.9024999999999999)
```

29.4.5 Root Finding Using Numpy

Here we'll use numpy to compute the roots of the characteristic polynomial

In [11]:
```
r1, r2 = np.roots([1, -ρ1, -ρ2])

p1 = cmath.polar(r1)
p2 = cmath.polar(r2)

print(f"r, ϕ = {r}, {ϕ}")
print(f"p1, p2 = {p1}, {p2}")
# print(f"g1, g2 = {g1}, {g2}")

print(f"a, b = {a}, {b}")
print(f"ρ1, ρ2 = {ρ1}, {ρ2}")
```

```
    r, ϕ = 0.95, 0.6283185307179586
p1, p2 = (0.95, 0.6283185307179586), (0.95, -0.6283185307179586)
a, b = (0.6346322893124001+0j), (0.9024999999999999-0j)
ρ1, ρ2 = 1.5371322893124, -0.9024999999999999
```

In [12]:
```
##=== This method uses numpy to calculate roots ===#


def y_nonstochastic(y_0=100, y_1=80, α=.9, β=.8, γ=10, n=80):
```

```
    """ Rather than computing the roots of the characteristic
    polynomial by hand as we did earlier, this function
    enlists numpy to do the work for us
    """

    # Useful constants
    ρ1 = α + β
    ρ2 = -β

    categorize_solution(ρ1, ρ2)

    # Find roots of polynomial
    roots = np.roots([1, -ρ1, -ρ2])
    print(f'Roots are {roots}')

    # Check if real or complex
    if all(isinstance(root, complex) for root in roots):
        print('Roots are complex')
    else:
        print('Roots are real')

    # Check if roots are less than one
    if all(abs(root) < 1 for root in roots):
        print('Roots are less than one')
    else:
        print('Roots are not less than one')

    # Define transition equation
    def transition(x, t): return ρ1 * x[t - 1] + ρ2 * x[t - 2] + γ

    # Set initial conditions
    y_t = [y_0, y_1]

    # Generate y_t series
    for t in range(2, n):
        y_t.append(transition(y_t, t))

    return y_t

plot_y(y_nonstochastic())
```

```
    Roots are complex with modulus less than one; therefore damped oscillations
Roots are [0.85+0.27838822j 0.85-0.27838822j]
Roots are complex
Roots are less than one
```

29.4.6 Reverse-Engineered Complex Roots: Example

The next cell studies the implications of reverse-engineered complex roots.

We'll generate an **undamped** cycle of period 10

```
In [13]: r = 1   # Generates undamped, nonexplosive cycles

         period = 10   # Length of cycle in units of time
         ϕ = 2 * math.pi/period

         ## Apply the reverse-engineering function f

         ρ1, ρ2, a, b = f(r, ϕ)

         # Drop the imaginary part so that it is a valid input into y_nonstochastic
         a = a.real
         b = b.real

         print(f"a, b = {a}, {b}")

         ytemp = y_nonstochastic(α=a, β=b, y_0=20, y_1=30)
         plot_y(ytemp)
```

```
a, b = 0.6180339887498949, 1.0
Roots are complex with modulus less than one; therefore damped oscillations
Roots are [0.80901699+0.58778525j 0.80901699-0.58778525j]
Roots are complex
Roots are less than one
```

29.4.7 Digression: Using Sympy to Find Roots

We can also use sympy to compute analytic formulas for the roots

In [14]:
```
init_printing()

r1 = Symbol("ρ_1")
r2 = Symbol("ρ_2")
z = Symbol("z")

sympy.solve(z**2 - r1*z - r2, z)
```

Out[14]:

$$\left[\frac{\rho_1}{2} - \frac{\sqrt{\rho_1^2 + 4\rho_2}}{2}, \quad \frac{\rho_1}{2} + \frac{\sqrt{\rho_1^2 + 4\rho_2}}{2}\right]$$

$$\left[\frac{\rho_1}{2} - \frac{1}{2}\sqrt{\rho_1^2 + 4\rho_2}, \quad \frac{\rho_1}{2} + \frac{1}{2}\sqrt{\rho_1^2 + 4\rho_2}\right]$$

In [15]:
```
a = Symbol("α")
b = Symbol("β")
r1 = a + b
r2 = -b

sympy.solve(z**2 - r1*z - r2, z)
```

Out[15]:

$$\left[\frac{\alpha}{2} + \frac{\beta}{2} - \frac{\sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}}{2}, \quad \frac{\alpha}{2} + \frac{\beta}{2} + \frac{\sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}}{2}\right]$$

$$\left[\frac{\alpha}{2} + \frac{\beta}{2} - \frac{1}{2}\sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}, \quad \frac{\alpha}{2} + \frac{\beta}{2} + \frac{1}{2}\sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}\right]$$

29.5 Stochastic Shocks

Now we'll construct some code to simulate the stochastic version of the model that emerges when we add a random shock process to aggregate demand

In [16]:
```
def y_stochastic(y_0=0, y_1=0, α=0.8, β=0.2, γ=10, n=100, σ=5):

    """This function takes parameters of a stochastic version of
    the model and proceeds to analyze the roots of the characteristic
    polynomial and also generate a simulation.
    """

    # Useful constants
    ρ1 = α + β
    ρ2 = -β

    # Categorize solution
    categorize_solution(ρ1, ρ2)

    # Find roots of polynomial
    roots = np.roots([1, -ρ1, -ρ2])
    print(roots)

    # Check if real or complex
    if all(isinstance(root, complex) for root in roots):
        print('Roots are complex')
    else:
        print('Roots are real')

    # Check if roots are less than one
    if all(abs(root) < 1 for root in roots):
        print('Roots are less than one')
    else:
        print('Roots are not less than one')

    # Generate shocks
    ϵ = np.random.normal(0, 1, n)

    # Define transition equation
    def transition(x, t): return ρ1 * \
        x[t - 1] + ρ2 * x[t - 2] + γ + σ * ϵ[t]

    # Set initial conditions
    y_t = [y_0, y_1]

    # Generate y_t series
    for t in range(2, n):
        y_t.append(transition(y_t, t))

    return y_t

plot_y(y_stochastic())
```

```
Roots are real and absolute values are less than one; therefore get smooth
↪convergence
to a steady state
[0.7236068 0.2763932]
```

```
Roots are real
Roots are less than one
```

Let's do a simulation in which there are shocks and the characteristic polynomial has complex roots

```
In [17]: r = .97

         period = 10   #  Length of cycle in units of time
         ϕ = 2 * math.pi/period

         ### Apply the  reverse-engineering function f

         ρ1, ρ2, a, b = f(r, ϕ)

         # Drop the imaginary part so that it is a valid input into y_nonstochastic
         a = a.real
         b = b.real

         print(f"a, b = {a}, {b}")
         plot_y(y_stochastic(y_0=40, y_1 = 42, α=a, β=b, σ=2, n=100))
```

```
a, b = 0.6285929690873979, 0.9409000000000001
Roots are complex with modulus less than one; therefore damped oscillations
[0.78474648+0.57015169j 0.78474648-0.57015169j]
Roots are complex
Roots are less than one
```

29.6 Government Spending

This function computes a response to either a permanent or one-off increase in government expenditures

```
In [18]: def y_stochastic_g(y_0=20,
                            y_1=20,
                            α=0.8,
                            β=0.2,
                            γ=10,
                            n=100,
                            σ=2,
                            g=0,
                            g_t=0,
                            duration='permanent'):

         """This program computes a response to a permanent increase
         in government expenditures that occurs at time 20
         """

         # Useful constants
         ρ1 = α + β
         ρ2 = -β

         # Categorize solution
         categorize_solution(ρ1, ρ2)

         # Find roots of polynomial
         roots = np.roots([1, -ρ1, -ρ2])
         print(roots)
```

```
# Check if real or complex
if all(isinstance(root, complex) for root in roots):
    print('Roots are complex')
else:
    print('Roots are real')

# Check if roots are less than one
if all(abs(root) < 1 for root in roots):
    print('Roots are less than one')
else:
    print('Roots are not less than one')

# Generate shocks
ϵ = np.random.normal(0, 1, n)

def transition(x, t, g):

    # Non-stochastic - separated to avoid generating random series
    # when not needed
    if σ == 0:
        return ρ1 * x[t - 1] + ρ2 * x[t - 2] + γ + g

    # Stochastic
    else:
        ϵ = np.random.normal(0, 1, n)
        return ρ1 * x[t - 1] + ρ2 * x[t - 2] + γ + g + σ * ϵ[t]

# Create list and set initial conditions
y_t = [y_0, y_1]

# Generate y_t series
for t in range(2, n):

    # No government spending
    if g == 0:
        y_t.append(transition(y_t, t))

    # Government spending (no shock)
    elif g != 0 and duration == None:
        y_t.append(transition(y_t, t))

    # Permanent government spending shock
    elif duration == 'permanent':
        if t < g_t:
            y_t.append(transition(y_t, t, g=0))
        else:
            y_t.append(transition(y_t, t, g=g))

    # One-off government spending shock
    elif duration == 'one-off':
        if t == g_t:
            y_t.append(transition(y_t, t, g=g))
        else:
            y_t.append(transition(y_t, t, g=0))
return y_t
```

A permanent government spending shock can be simulated as follows

```
In [19]: plot_y(y_stochastic_g(g=10, g_t=20, duration='permanent'))
```

```
    Roots are real and absolute values are less than one; therefore get smooth
 ↪convergence
to a steady state
[0.7236068 0.2763932]
Roots are real
Roots are less than one
```

We can also see the response to a one time jump in government expenditures

```
In [20]: plot_y(y_stochastic_g(g=500, g_t=50, duration='one-off'))
```

```
    Roots are real and absolute values are less than one; therefore get smooth
 ↪convergence
to a steady state
[0.7236068 0.2763932]
Roots are real
Roots are less than one
```

29.7 Wrapping Everything Into a Class

Up to now, we have written functions to do the work.

Now we'll roll up our sleeves and write a Python class called `Samuelson` for the Samuelson model

```
In [21]: class Samuelson():

             """This class represents the Samuelson model, otherwise known as the
            multiple-accelerator model. The model combines the Keynesian multiplier
             with the accelerator theory of investment.

             The path of output is governed by a linear second-order difference
 ↪equation

             .. math::

                 Y_t =  + \alpha (1 + \beta) Y_{t-1} - \alpha \beta Y_{t-2}

             Parameters
             ----------
             y_0 : scalar
                 Initial condition for Y_0
             y_1 : scalar
                 Initial condition for Y_1
             α : scalar
                 Marginal propensity to consume
             β : scalar
                 Accelerator coefficient
             n : int
```

```
        Number of iterations
    σ : scalar
        Volatility parameter. It must be greater than or equal to 0. Set
        equal to 0 for a non-stochastic model.
    g : scalar
        Government spending shock
    g_t : int
        Time at which government spending shock occurs. Must be specified
        when duration != None.
    duration : {None, 'permanent', 'one-off'}
        Specifies type of government spending shock. If none, government
        spending equal to g for all t.

    """

    def __init__(self,
                 y_0=100,
                 y_1=50,
                 α=1.3,
                 β=0.2,
                 γ=10,
                 n=100,
                 σ=0,
                 g=0,
                 g_t=0,
                 duration=None):

        self.y_0, self.y_1, self.α, self.β = y_0, y_1, α, β
        self.n, self.g, self.g_t, self.duration = n, g, g_t, duration
        self.γ, self.σ = γ, σ
        self.ρ1 = α + β
        self.ρ2 = -β
        self.roots = np.roots([1, -self.ρ1, -self.ρ2])

    def root_type(self):
        if all(isinstance(root, complex) for root in self.roots):
            return 'Complex conjugate'
        elif len(self.roots) > 1:
            return 'Double real'
        else:
            return 'Single real'

    def root_less_than_one(self):
        if all(abs(root) < 1 for root in self.roots):
            return True

    def solution_type(self):
        ρ1, ρ2 = self.ρ1, self.ρ2
        discriminant = ρ1 ** 2 + 4 * ρ2
        if ρ2 >= 1 + ρ1 or ρ2 <= -1:
            return 'Explosive oscillations'
        elif ρ1 + ρ2 >= 1:
            return 'Explosive growth'
        elif discriminant < 0:
            return 'Damped oscillations'
        else:
            return 'Steady state'
```

```
    def _transition(self, x, t, g):

        # Non-stochastic - separated to avoid generating random series
        # when not needed
        if self.σ == 0:
            return self.ρ1 * x[t - 1] + self.ρ2 * x[t - 2] + self.γ + g

        # Stochastic
        else:
            ϵ = np.random.normal(0, 1, self.n)
            return self.ρ1 * x[t - 1] + self.ρ2 * x[t - 2] + self.γ + g \
                + self.σ * ϵ[t]

    def generate_series(self):

        # Create list and set initial conditions
        y_t = [self.y_0, self.y_1]

        # Generate y_t series
        for t in range(2, self.n):

            # No government spending
            if self.g == 0:
                y_t.append(self._transition(y_t, t))

            # Government spending (no shock)
            elif self.g != 0 and self.duration == None:
                y_t.append(self._transition(y_t, t))

            # Permanent government spending shock
            elif self.duration == 'permanent':
                if t < self.g_t:
                    y_t.append(self._transition(y_t, t, g=0))
                else:
                    y_t.append(self._transition(y_t, t, g=self.g))

            # One-off government spending shock
            elif self.duration == 'one-off':
                if t == self.g_t:
                    y_t.append(self._transition(y_t, t, g=self.g))
                else:
                    y_t.append(self._transition(y_t, t, g=0))
        return y_t

    def summary(self):
        print('Summary\n' + '-' * 50)
        print(f'Root type: {self.root_type()}')
        print(f'Solution type: {self.solution_type()}')
        print(f'Roots: {str(self.roots)}')

        if self.root_less_than_one() == True:
            print('Absolute value of roots is less than one')
        else:
            print('Absolute value of roots is not less than one')

        if self.σ > 0:
            print('Stochastic series with σ = ' + str(self.σ))
        else:
```

```
            print('Non-stochastic series')

        if self.g != 0:
            print('Government spending equal to ' + str(self.g))

        if self.duration != None:
            print(self.duration.capitalize() +
                  ' government spending shock at t = ' + str(self.g_t))

    def plot(self):
        fig, ax = plt.subplots(figsize=(10, 6))
        ax.plot(self.generate_series())
        ax.set(xlabel='Iteration', xlim=(0, self.n))
        ax.set_ylabel('$Y_t$', rotation=0)
        ax.grid()

        # Add parameter values to plot
        paramstr = f'$\\alpha={self.α:.2f}$ \n $\\beta={self.β:.2f}$ \n \
        $\\gamma={self.γ:.2f}$ \n $\\sigma={self.σ:.2f}$ \n \
        $\\rho_1={self.ρ1:.2f}$ \n $\\rho_2={self.ρ2:.2f}$'
        props = dict(fc='white', pad=10, alpha=0.5)
        ax.text(0.87, 0.05, paramstr, transform=ax.transAxes,
                fontsize=12, bbox=props, va='bottom')

        return fig

    def param_plot(self):

        # Uses the param_plot() function defined earlier (it is then able
        # to be used standalone or as part of the model)

        fig = param_plot()
        ax = fig.gca()

        # Add λ values to legend
        for i, root in enumerate(self.roots):
            if isinstance(root, complex):
              # Need to fill operator for positive as string is split apart
                operator = ['+', '']
                label = rf'$\lambda_{i+1} = {sam.roots[i].real:.2f} \
                    {operator[i]} {sam.roots[i].imag:.2f}i$'
            else:
                label = rf'$\lambda_{i+1} = {sam.roots[i].real:.2f}$'
            ax.scatter(0, 0, 0, label=label) # dummy to add to legend

        # Add ϱ pair to plot
        ax.scatter(self.ρ1, self.ρ2, 100, 'red', '+',
            label=r'$(\ \rho_1, \ \rho_2 \ )$', zorder=5)

        plt.legend(fontsize=12, loc=3)

        return fig
```

29.7.1 Illustration of Samuelson Class

Now we'll put our Samuelson class to work on an example

```
In [22]: sam = Samuelson(α=0.8, β=0.5, σ=2, g=10, g_t=20, duration='permanent')
         sam.summary()
```

```
    Summary
--------------------------------------------------
Root type: Complex conjugate
Solution type: Damped oscillations
Roots: [0.65+0.27838822j 0.65-0.27838822j]
Absolute value of roots is less than one
Stochastic series with σ = 2
Government spending equal to 10
Permanent government spending shock at t = 20
```

```
In [23]: sam.plot()
         plt.show()
```

29.7.2 Using the Graph

We'll use our graph to show where the roots lie and how their location is consistent with the behavior of the path just graphed.

The red + sign shows the location of the roots

```
In [24]: sam.param_plot()
         plt.show()
```

29.8 Using the LinearStateSpace Class

It turns out that we can use the QuantEcon.py LinearStateSpace class to do much of the work that we have done from scratch above.

Here is how we map the Samuelson model into an instance of a `LinearStateSpace` class

In [25]:
```
"""This script maps the Samuelson model in the the
``LinearStateSpace`` class
"""
α = 0.8
β = 0.9
ρ1 = α + β
ρ2 = -β
γ = 10
σ = 1
g = 10
n = 100

A = [[1,         0,       0],
     [γ + g,    ρ1,      ρ2],
     [0,         1,       0]]

G = [[γ + g, ρ1,    ρ2],          # this is Y_{t+1}
     [γ,       α,     0],          # this is C_{t+1}
     [0,       β,    -β]]          # this is I_{t+1}

μ_0 = [1, 100, 100]
C = np.zeros((3,1))
```

```
C[1] = σ # stochastic

sam_t = LinearStateSpace(A, C, G, mu_0=μ_0)

x, y = sam_t.simulate(ts_length=n)

fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
titles = ['Output ($Y_t$)', 'Consumption ($C_t$)', 'Investment ($I_t$)']
colors = ['darkblue', 'red', 'purple']
for ax, series, title, color in zip(axes, y, titles, colors):
    ax.plot(series, color=color)
    ax.set(title=title, xlim=(0, n))
    ax.grid()

axes[-1].set_xlabel('Iteration')

plt.show()
```

29.8.1 Other Methods in the `LinearStateSpace` Class

Let's plot **impulse response functions** for the instance of the Samuelson model using a method in the `LinearStateSpace` class

```
In [26]: imres = sam_t.impulse_response()
         imres = np.asarray(imres)
         y1 = imres[:, :, 0]
         y2 = imres[:, :, 1]
         y1.shape
```

Out[26]: (2, 6, 1)

$$(2, \quad 6, \quad 1)$$

Now let's compute the zeros of the characteristic polynomial by simply calculating the eigenvalues of A

In [27]:
```
A = np.asarray(A)
w, v = np.linalg.eig(A)
print(w)
```

```
[0.85+0.42130749j 0.85-0.42130749j 1.  +0.j        ]
```

29.8.2 Inheriting Methods from `LinearStateSpace`

We could also create a subclass of `LinearStateSpace` (inheriting all its methods and attributes) to add more functions to use

In [28]:
```
class SamuelsonLSS(LinearStateSpace):

    """
    This subclass creates a Samuelson multiplier-accelerator model
    as a linear state space system.
    """
    def __init__(self,
                 y_0=100,
                 y_1=100,
                 α=0.8,
                 β=0.9,
                 γ=10,
                 σ=1,
                 g=10):

        self.α, self.β = α, β
        self.y_0, self.y_1, self.g = y_0, y_1, g
        self.γ, self.σ = γ, σ

        # Define intial conditions
        self.μ_0 = [1, y_0, y_1]

        self.ρ1 = α + β
        self.ρ2 = -β

        # Define transition matrix
        self.A = [[1,                  0,          0],
                  [γ + g,        self.ρ1,    self.ρ2],
                  [0,                  1,          0]]

        # Define output matrix
        self.G = [[γ + g, self.ρ1, self.ρ2],          # this is Y_{t+1}
                  [γ,            α,        0],          # this is C_{t+1}
                  [0,            β,       -β]]          # this is I_{t+1}
```

```
        self.C = np.zeros((3, 1))
        self.C[1] = σ  # stochastic

        # Initialize LSS with parameters from Samuelson model
        LinearStateSpace.__init__(self, self.A, self.C, self.G, mu_0=self.
↪μ_0)

    def plot_simulation(self, ts_length=100, stationary=True):

        # Temporarily store original parameters
        temp_μ = self.μ_0
        temp_Σ = self.Sigma_0

        # Set distribution parameters equal to their stationary
        # values for simulation
        if stationary == True:
            try:
                self.μ_x, self.μ_y, self.σ_x, self.σ_y = \
                    self.stationary_distributions()
                self.μ_0 = self.μ_y
                self.Σ_0 = self.σ_y
            # Exception where no convergence achieved when
            #calculating stationary distributions
            except ValueError:
                print('Stationary distribution does not exist')

        x, y = self.simulate(ts_length)

        fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
        titles = ['Output ($Y_t$)', 'Consumption ($C_t$)', 'Investment
↪($I_t$)']
        colors = ['darkblue', 'red', 'purple']
        for ax, series, title, color in zip(axes, y, titles, colors):
            ax.plot(series, color=color)
            ax.set(title=title, xlim=(0, n))
            ax.grid()

        axes[-1].set_xlabel('Iteration')

        # Reset distribution parameters to their initial values
        self.μ_0 = temp_μ
        self.Sigma_0 = temp_Σ

        return fig

    def plot_irf(self, j=5):

        x, y = self.impulse_response(j)

        # Reshape into 3 x j matrix for plotting purposes
        yimf = np.array(y).flatten().reshape(j+1, 3).T

        fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
        labels = ['$Y_t$', '$C_t$', '$I_t$']
        colors = ['darkblue', 'red', 'purple']
        for ax, series, label, color in zip(axes, yimf, labels, colors):
            ax.plot(series, color=color)
            ax.set(xlim=(0, j))
```

```
            ax.set_ylabel(label, rotation=0, fontsize=14, labelpad=10)
            ax.grid()

        axes[0].set_title('Impulse Response Functions')
        axes[-1].set_xlabel('Iteration')

        return fig

    def multipliers(self, j=5):
        x, y = self.impulse_response(j)
        return np.sum(np.array(y).flatten().reshape(j+1, 3), axis=0)
```

29.8.3 Illustrations

Let's show how we can use the `SamuelsonLSS`

```
In [29]: samlss = SamuelsonLSS()
```

```
In [30]: samlss.plot_simulation(100, stationary=False)
         plt.show()
```

```
In [31]: samlss.plot_simulation(100, stationary=True)
         plt.show()
```

```
In [32]: samlss.plot_irf(100)
         plt.show()
```

```
In [33]: samlss.multipliers()
```

```
Out[33]: array([7.414389, 6.835896, 0.578493])
```

29.9 Pure Multiplier Model

Let's shut down the accelerator by setting $b = 0$ to get a pure multiplier model

- the absence of cycles gives an idea about why Samuelson included the accelerator

```
In [34]: pure_multiplier = SamuelsonLSS(α=0.95, β=0)
```

```
In [35]: pure_multiplier.plot_simulation()
```

```
Stationary distribution does not exist
```

[35]:

```
In [36]: pure_multiplier = SamuelsonLSS(α=0.8, β=0)
```

```
In [37]: pure_multiplier.plot_simulation()
```

[37]:

```
In [38]: pure_multiplier.plot_irf(100)
```

[38]:

29.10 Summary

In this lecture, we wrote functions and classes to represent non-stochastic and stochastic versions of the Samuelson (1939) multiplier-accelerator model, described in [139].

We saw that different parameter values led to different output paths, which could either be stationary, explosive, or oscillating.

We also were able to represent the model using the QuantEcon.py LinearStateSpace class.

Chapter 30

Kesten Processes and Firm Dynamics

30.1 Contents

- Overview 30.2
- Kesten Processes 30.3
- Heavy Tails 30.4
- Application: Firm Dynamics 30.5
- Exercises 30.6
- Solutions 30.7

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

```
In [1]: !pip install --upgrade quantecon
        !pip install --upgrade yfinance
```

30.2 Overview

Previously we learned about linear scalar-valued stochastic processes (AR(1) models).

Now we generalize these linear models slightly by allowing the multiplicative coefficient to be stochastic.

Such processes are known as Kesten processes after German–American mathematician Harry Kesten (1931–2019)

Although simple to write down, Kesten processes are interesting for at least two reasons:

1. A number of significant economic processes are or can be described as Kesten processes.
2. Kesten processes generate interesting dynamics, including, in some cases, heavy-tailed cross-sectional distributions.

We will discuss these issues as we go along.

Let's start with some imports:

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import quantecon as qe
```

The following two lines are only added to avoid a `FutureWarning` caused by compatibility issues between pandas and matplotlib.

In [3]:
```
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

Additional technical background related to this lecture can be found in the monograph of [27].

30.3 Kesten Processes

A **Kesten process** is a stochastic process of the form

$$X_{t+1} = a_{t+1} X_t + \eta_{t+1} \tag{1}$$

where $\{a_t\}_{t \geq 1}$ and $\{\eta_t\}_{t \geq 1}$ are IID sequences.

We are interested in the dynamics of $\{X_t\}_{t \geq 0}$ when X_0 is given.

We will focus on the nonnegative scalar case, where X_t takes values in $\mathbb{R}_+$.

In particular, we will assume that

- the initial condition X_0 is nonnegative,
- $\{a_t\}_{t \geq 1}$ is a nonnegative IID stochastic process and
- $\{\eta_t\}_{t \geq 1}$ is another nonnegative IID stochastic process, independent of the first.

30.3.1 Example: GARCH Volatility

The GARCH model is common in financial applications, where time series such as asset returns exhibit time varying volatility.

For example, consider the following plot of daily returns on the Nasdaq Composite Index for the period 1st January 2006 to 1st November 2019.

In [4]:
```
import yfinance as yf
import pandas as pd

s = yf.download('^IXIC', '2006-1-1', '2019-11-1')['Adj Close']

r = s.pct_change()

fig, ax = plt.subplots()

ax.plot(r, alpha=0.7)

ax.set_ylabel('returns', fontsize=12)
```

```
ax.set_xlabel('date', fontsize=12)

plt.show()
```

```
[*********************100%***********************]  1 of 1 completed
```

Notice how the series exhibits bursts of volatility (high variance) and then settles down again.

GARCH models can replicate this feature.

The GARCH(1, 1) volatility process takes the form

$$\sigma_{t+1}^2 = \alpha_0 + \sigma_t^2(\alpha_1 \xi_{t+1}^2 + \beta) \tag{2}$$

where $\{\xi_t\}$ is IID with $\mathbb{E}\xi_t^2 = 1$ and all parameters are positive.

Returns on a given asset are then modeled as

$$r_t = \sigma_t \zeta_t \tag{3}$$

where $\{\zeta_t\}$ is again IID and independent of $\{\xi_t\}$.

The volatility sequence $\{\sigma_t^2\}$, which drives the dynamics of returns, is a Kesten process.

30.3.2 Example: Wealth Dynamics

Suppose that a given household saves a fixed fraction s of its current wealth in every period.

The household earns labor income y_t at the start of time t.

Wealth then evolves according to

$$w_{t+1} = R_{t+1} s w_t + y_{t+1} \tag{4}$$

where $\{R_t\}$ is the gross rate of return on assets.

If $\{R_t\}$ and $\{y_t\}$ are both IID, then (4) is a Kesten process.

30.3.3 Stationarity

In earlier lectures, such as the one on AR(1) processes, we introduced the notion of a stationary distribution.

In the present context, we can define a stationary distribution as follows:

The distribution F^* on $\mathbb{R}$ is called **stationary** for the Kesten process (1) if

$$X_t \sim F^* \quad \Longrightarrow \quad a_{t+1} X_t + \eta_{t+1} \sim F^* \tag{5}$$

In other words, if the current state X_t has distribution F^*, then so does the next period state X_{t+1}.

We can write this alternatively as

$$F^*(y) = \int \mathbb{P}\{a_{t+1} x + \eta_{t+1} \leq y\} F^*(dx) \quad \text{for all } y \geq 0. \tag{6}$$

The left hand side is the distribution of the next period state when the current state is drawn from F^*.

The equality in (6) states that this distribution is unchanged.

30.3.4 Cross-Sectional Interpretation

There is an important cross-sectional interpretation of stationary distributions, discussed previously but worth repeating here.

Suppose, for example, that we are interested in the wealth distribution — that is, the current distribution of wealth across households in a given country.

Suppose further that

- the wealth of each household evolves independently according to (4),
- F^* is a stationary distribution for this stochastic process and
- there are many households.

Then F^* is a steady state for the cross-sectional wealth distribution in this country.

In other words, if F^* is the current wealth distribution then it will remain so in subsequent periods, *ceteris paribus*.

To see this, suppose that F^* is the current wealth distribution.

What is the fraction of households with wealth less than y next period?

To obtain this, we sum the probability that wealth is less than y tomorrow, given that current wealth is w, weighted by the fraction of households with wealth w.

Noting that the fraction of households with wealth in interval dw is $F^*(dw)$, we get

$$\int \mathbb{P}\{R_{t+1}sw + y_{t+1} \leq y\}F^*(dw)$$

By the definition of stationarity and the assumption that F^* is stationary for the wealth process, this is just $F^*(y)$.

Hence the fraction of households with wealth in $[0, y]$ is the same next period as it is this period.

Since y was chosen arbitrarily, the distribution is unchanged.

30.3.5 Conditions for Stationarity

The Kesten process $X_{t+1} = a_{t+1}X_t + \eta_{t+1}$ does not always have a stationary distribution.

For example, if $a_t \equiv \eta_t \equiv 1$ for all t, then $X_t = X_0 + t$, which diverges to infinity.

To prevent this kind of divergence, we require that $\{a_t\}$ is strictly less than 1 most of the time.

In particular, if

$$\mathbb{E} \ln a_t < 0 \quad \text{and} \quad \mathbb{E}\eta_t < \infty \tag{7}$$

then a unique stationary distribution exists on $\mathbb{R}_+$.

- See, for example, theorem 2.1.3 of [27], which provides slightly weaker conditions.

As one application of this result, we see that the wealth process (4) will have a unique stationary distribution whenever labor income has finite mean and $\mathbb{E} \ln R_t + \ln s < 0$.

30.4 Heavy Tails

Under certain conditions, the stationary distribution of a Kesten process has a Pareto tail.

(See our earlier lecture on heavy-tailed distributions for background.)

This fact is significant for economics because of the prevalence of Pareto-tailed distributions.

30.4.1 The Kesten–Goldie Theorem

To state the conditions under which the stationary distribution of a Kesten process has a Pareto tail, we first recall that a random variable is called **nonarithmetic** if its distribution is not concentrated on $\{\dots, -2t, -t, 0, t, 2t, \dots\}$ for any $t \geq 0$.

For example, any random variable with a density is nonarithmetic.

The famous Kesten–Goldie Theorem (see, e.g., [27], theorem 2.4.4) states that if

1. the stationarity conditions in (7) hold,
2. the random variable a_t is positive with probability one and nonarithmetic,

3. $\mathbb{P}\{a_t x + \eta_t = x\} < 1$ for all $x \in \mathbb{R}_+$ and
4. there exists a positive constant α such that

$$\mathbb{E}a_t^\alpha = 1, \quad \mathbb{E}\eta_t^\alpha < \infty, \quad \text{and} \quad \mathbb{E}[a_t^{\alpha+1}] < \infty$$

then the stationary distribution of the Kesten process has a Pareto tail with tail index α.

More precisely, if F^* is the unique stationary distribution and $X^* \sim F^*$, then

$$\lim_{x\to\infty} x^\alpha \mathbb{P}\{X^* > x\} = c$$

for some positive constant c.

30.4.2 Intuition

Later we will illustrate the Kesten–Goldie Theorem using rank-size plots.

Prior to doing so, we can give the following intuition for the conditions.

Two important conditions are that $\mathbb{E}\ln a_t < 0$, so the model is stationary, and $\mathbb{E}a_t^\alpha = 1$ for some $\alpha > 0$.

The first condition implies that the distribution of a_t has a large amount of probability mass below 1.

The second condition implies that the distribution of a_t has at least some probability mass at or above 1.

The first condition gives us existence of the stationary condition.

The second condition means that the current state can be expanded by a_t.

If this occurs for several concurrent periods, the effects compound each other, since a_t is multiplicative.

This leads to spikes in the time series, which fill out the extreme right hand tail of the distribution.

The spikes in the time series are visible in the following simulation, which generates of 10 paths when a_t and b_t are lognormal.

In [5]:
```
μ = -0.5
σ = 1.0

def kesten_ts(ts_length=100):
    x = np.zeros(ts_length)
    for t in range(ts_length-1):
        a = np.exp(μ + σ * np.random.randn())
        b = np.exp(np.random.randn())
        x[t+1] = a * x[t] + b
    return x

fig, ax = plt.subplots()

num_paths = 10
```

```
np.random.seed(12)

for i in range(num_paths):
    ax.plot(kesten_ts())

ax.set(xlabel='time', ylabel='$X_t$')
plt.show()
```

30.5 Application: Firm Dynamics

As noted in our lecture on heavy tails, for common measures of firm size such as revenue or employment, the US firm size distribution exhibits a Pareto tail (see, e.g., [13], [59]).

Let us try to explain this rather striking fact using the Kesten–Goldie Theorem.

30.5.1 Gibrat's Law

It was postulated many years ago by Robert Gibrat [62] that firm size evolves according to a simple rule whereby size next period is proportional to current size.

This is now known as Gibrat's law of proportional growth.

We can express this idea by stating that a suitably defined measure s_t of firm size obeys

$$\frac{s_{t+1}}{s_t} = a_{t+1} \tag{8}$$

for some positive IID sequence $\{a_t\}$.

One implication of Gibrat's law is that the growth rate of individual firms does not depend on their size.

However, over the last few decades, research contradicting Gibrat's law has accumulated in the literature.

For example, it is commonly found that, on average,

1. small firms grow faster than large firms (see, e.g., [53] and [66]) and
2. the growth rate of small firms is more volatile than that of large firms [50].

On the other hand, Gibrat's law is generally found to be a reasonable approximation for large firms [53].

We can accommodate these empirical findings by modifying (8) to

$$s_{t+1} = a_{t+1} s_t + b_{t+1} \tag{9}$$

where $\{a_t\}$ and $\{b_t\}$ are both IID and independent of each other.

In the exercises you are asked to show that (9) is more consistent with the empirical findings presented above than Gibrat's law in (8).

30.5.2 Heavy Tails

So what has this to do with Pareto tails?

The answer is that (9) is a Kesten process.

If the conditions of the Kesten–Goldie Theorem are satisfied, then the firm size distribution is predicted to have heavy tails — which is exactly what we see in the data.

In the exercises below we explore this idea further, generalizing the firm size dynamics and examining the corresponding rank-size plots.

We also try to illustrate why the Pareto tail finding is significant for quantitative analysis.

30.6 Exercises

30.6.1 Exercise 1

Simulate and plot 15 years of daily returns (consider each year as having 250 working days) using the GARCH(1, 1) process in (2)–(3).

Take ξ_t and ζ_t to be independent and standard normal.

Set $\alpha_0 = 0.00001, \alpha_1 = 0.1, \beta = 0.9$ and $\sigma_0 = 0$.

Compare visually with the Nasdaq Composite Index returns shown above.

While the time path differs, you should see bursts of high volatility.

30.6.2 Exercise 2

In our discussion of firm dynamics, it was claimed that (9) is more consistent with the empirical literature than Gibrat's law in (8).

(The empirical literature was reviewed immediately above (9).)

In what sense is this true (or false)?

30.6.3 Exercise 3

Consider an arbitrary Kesten process as given in (1).

Suppose that $\{a_t\}$ is lognormal with parameters (μ, σ).

In other words, each a_t has the same distribution as $\exp(\mu + \sigma Z)$ when Z is standard normal.

Suppose further that $\mathbb{E}\eta_t^r < \infty$ for every $r > 0$, as would be the case if, say, η_t is also lognormal.

Show that the conditions of the Kesten–Goldie theorem are satisfied if and only if $\mu < 0$.

Obtain the value of α that makes the Kesten–Goldie conditions hold.

30.6.4 Exercise 4

One unrealistic aspect of the firm dynamics specified in (9) is that it ignores entry and exit.

In any given period and in any given market, we observe significant numbers of firms entering and exiting the market.

Empirical discussion of this can be found in a famous paper by Hugo Hopenhayn [85].

In the same paper, Hopenhayn builds a model of entry and exit that incorporates profit maximization by firms and market clearing quantities, wages and prices.

In his model, a stationary equilibrium occurs when the number of entrants equals the number of exiting firms.

In this setting, firm dynamics can be expressed as

$$s_{t+1} = e_{t+1}\mathbb{1}\{s_t < \bar{s}\} + (a_{t+1}s_t + b_{t+1})\mathbb{1}\{s_t \geq \bar{s}\} \tag{10}$$

Here

- the state variable s_t is represents productivity (which is a proxy for output and hence firm size),
- the IID sequence $\{e_t\}$ is thought of as a productivity draw for a new entrant and
- the variable $\bar{s}$ is a threshold value that we take as given, although it is determined endogenously in Hopenhayn's model.

The idea behind (10) is that firms stay in the market as long as their productivity s_t remains at or above $\bar{s}$.

- In this case, their productivity updates according to (9).

Firms choose to exit when their productivity s_t falls below $\bar{s}$.

- In this case, they are replaced by a new firm with productivity e_{t+1}.

What can we say about dynamics?

Although (10) is not a Kesten process, it does update in the same way as a Kesten process when s_t is large.

So perhaps its stationary distribution still has Pareto tails?

Your task is to investigate this question via simulation and rank-size plots.

The approach will be to

1. generate M draws of s_T when M and T are large and
2. plot the largest 1,000 of the resulting draws in a rank-size plot.

(The distribution of s_T will be close to the stationary distribution when T is large.)

In the simulation, assume that

- each of a_t, b_t and e_t is lognormal,
- the parameters are

In [6]:
```
μ_a = -0.5          # location parameter for a
σ_a = 0.1           # scale parameter for a
μ_b = 0.0           # location parameter for b
σ_b = 0.5           # scale parameter for b
μ_e = 0.0           # location parameter for e
σ_e = 0.5           # scale parameter for e
s_bar = 1.0         # threshold
T = 500             # sampling date
M = 1_000_000       # number of firms
s_init = 1.0        # initial condition for each firm
```

30.7 Solutions

30.7.1 Exercise 1

Here is one solution:

In [7]:
```
α_0 = 1e-5
α_1 = 0.1
β = 0.9

years = 15
days = years * 250

def garch_ts(ts_length=days):
    σ2 = 0
    r = np.zeros(ts_length)
    for t in range(ts_length-1):
        ξ = np.random.randn()
        σ2 = α_0 + σ2 * (α_1 * ξ**2 + β)
        r[t] = np.sqrt(σ2) * np.random.randn()
    return r

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

```
np.random.seed(12)

ax.plot(garch_ts(), alpha=0.7)

ax.set(xlabel='time', ylabel='$\\sigma_t^2$')
plt.show()
```

30.7.2 Exercise 2

The empirical findings are that

1. small firms grow faster than large firms and
2. the growth rate of small firms is more volatile than that of large firms.

Also, Gibrat's law is generally found to be a reasonable approximation for large firms than for small firms

The claim is that the dynamics in (9) are more consistent with points 1-2 than Gibrat's law.

To see why, we rewrite (9) in terms of growth dynamics:

$$\frac{s_{t+1}}{s_t} = a_{t+1} + \frac{b_{t+1}}{s_t} \tag{11}$$

Taking $s_t = s$ as given, the mean and variance of firm growth are

$$\mathbb{E}a + \frac{\mathbb{E}b}{s} \quad \text{and} \quad \mathbb{V}a + \frac{\mathbb{V}b}{s^2}$$

Both of these decline with firm size s, consistent with the data.

Moreover, the law of motion (11) clearly approaches Gibrat's law (8) as s_t gets large.

30.7.3 Exercise 3

Since a_t has a density it is nonarithmetic.

Since a_t has the same density as $a = \exp(\mu + \sigma Z)$ when Z is standard normal, we have

$$\mathbb{E} \ln a_t = \mathbb{E}(\mu + \sigma Z) = \mu,$$

and since η_t has finite moments of all orders, the stationarity condition holds if and only if $\mu < 0$.

Given the properties of the lognormal distribution (which has finite moments of all orders), the only other condition in doubt is existence of a positive constant α such that $\mathbb{E} a_t^\alpha = 1$.

This is equivalent to the statement

$$\exp\left(\alpha\mu + \frac{\alpha^2\sigma^2}{2}\right) = 1.$$

Solving for α gives $\alpha = -2\mu/\sigma^2$.

30.7.4 Exercise 4

Here's one solution. First we generate the observations:

In [8]:
```
from numba import njit, prange
from numpy.random import randn


@njit(parallel=True)
def generate_draws(μ_a=-0.5,
                   σ_a=0.1,
                   μ_b=0.0,
                   σ_b=0.5,
                   μ_e=0.0,
                   σ_e=0.5,
                   s_bar=1.0,
                   T=500,
                   M=1_000_000,
                   s_init=1.0):

    draws = np.empty(M)
    for m in prange(M):
        s = s_init
        for t in range(T):
            if s < s_bar:
                new_s = np.exp(μ_e + σ_e *  randn())
            else:
                a = np.exp(μ_a + σ_a * randn())
                b = np.exp(μ_b + σ_b * randn())
```

```
                new_s = a * s + b
            s = new_s
        draws[m] = s

    return draws

data = generate_draws()
```

Now we produce the rank-size plot:

```
In [9]: fig, ax = plt.subplots()

        qe.rank_size_plot(data, ax, c=0.01)

        plt.show()
```

The plot produces a straight line, consistent with a Pareto tail.

Chapter 31

Wealth Distribution Dynamics

31.1 Contents

- Overview 31.2
- Lorenz Curves and the Gini Coefficient 31.3
- A Model of Wealth Dynamics 31.4
- Implementation 31.5
- Applications 31.6
- Exercises 31.7
- Solutions 31.8

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

```
In [1]: !pip install --upgrade quantecon
```

31.2 Overview

This notebook gives an introduction to wealth distribution dynamics, with a focus on

- modeling and computing the wealth distribution via simulation,
- measures of inequality such as the Lorenz curve and Gini coefficient, and
- how inequality is affected by the properties of wage income and returns on assets.

One interesting property of the wealth distribution we discuss is Pareto tails.

The wealth distribution in many countries exhibits a Pareto tail

- See this lecture for a definition.
- For a review of the empirical evidence, see, for example, [17].

This is consistent with high concentration of wealth amongst the richest households.

It also gives us a way to quantify such concentration, in terms of the tail index.

One question of interest is whether or not we can replicate Pareto tails from a relatively simple model.

31.2.1 A Note on Assumptions

The evolution of wealth for any given household depends on their savings behavior.

Modeling such behavior will form an important part of this lecture series.

However, in this particular lecture, we will be content with rather ad hoc (but plausible) savings rules.

We do this to more easily explore the implications of different specifications of income dynamics and investment returns.

At the same time, all of the techniques discussed here can be plugged into models that use optimization to obtain savings rules.

We will use the following imports.

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import quantecon as qe
from numba import njit, jitclass, float64, prange
```

31.3 Lorenz Curves and the Gini Coefficient

Before we investigate wealth dynamics, we briefly review some measures of inequality.

31.3.1 Lorenz Curves

One popular graphical measure of inequality is the Lorenz curve.

The package QuantEcon.py, already imported above, contains a function to compute Lorenz curves.

To illustrate, suppose that

In [3]:
```
n = 10_000                        # size of sample
w = np.exp(np.random.randn(n))    # lognormal draws
```

is data representing the wealth of 10,000 households.

We can compute and plot the Lorenz curve as follows:

In [4]:
```
f_vals, l_vals = qe.lorenz_curve(w)

fig, ax = plt.subplots()
ax.plot(f_vals, l_vals, label='Lorenz curve, lognormal sample')
ax.plot(f_vals, f_vals, label='Lorenz curve, equality')
ax.legend()
plt.show()
```

This curve can be understood as follows: if point (x, y) lies on the curve, it means that, collectively, the bottom $(100x)\%$ of the population holds $(100y)\%$ of the wealth.

The "equality" line is the 45 degree line (which might not be exactly 45 degrees in the figure, depending on the aspect ratio).

A sample that produces this line exhibits perfect equality.

The other line in the figure is the Lorenz curve for the lognormal sample, which deviates significantly from perfect equality.

For example, the bottom 80% of the population holds around 40% of total wealth.

Here is another example, which shows how the Lorenz curve shifts as the underlying distribution changes.

We generate 10,000 observations using the Pareto distribution with a range of parameters, and then compute the Lorenz curve corresponding to each set of observations.

```
In [5]: a_vals = (1, 2, 5)              # Pareto tail index
        n = 10_000                      # size of each sample
        fig, ax = plt.subplots()
        for a in a_vals:
            u = np.random.uniform(size=n)
            y = u**(-1/a)               # distributed as Pareto with tail index a
            f_vals, l_vals = qe.lorenz_curve(y)
            ax.plot(f_vals, l_vals, label=f'$a = {a}$')
        ax.plot(f_vals, f_vals, label='equality')
        ax.legend()
        plt.show()
```

You can see that, as the tail parameter of the Pareto distribution increases, inequality decreases.

This is to be expected, because a higher tail index implies less weight in the tail of the Pareto distribution.

31.3.2 The Gini Coefficient

The definition and interpretation of the Gini coefficient can be found on the corresponding Wikipedia page.

A value of 0 indicates perfect equality (corresponding the case where the Lorenz curve matches the 45 degree line) and a value of 1 indicates complete inequality (all wealth held by the richest household).

The QuantEcon.py library contains a function to calculate the Gini coefficient.

We can test it on the Weibull distribution with parameter a, where the Gini coefficient is known to be

$$G = 1 - 2^{-1/a}$$

Let's see if the Gini coefficient computed from a simulated sample matches this at each fixed value of a.

```
In [6]: a_vals = range(1, 20)
        ginis = []
        ginis_theoretical = []
        n = 100

        fig, ax = plt.subplots()
        for a in a_vals:
```

```
    y = np.random.weibull(a, size=n)
    ginis.append(qe.gini_coefficient(y))
    ginis_theoretical.append(1 - 2**(-1/a))
ax.plot(a_vals, ginis, label='estimated gini coefficient')
ax.plot(a_vals, ginis_theoretical, label='theoretical gini coefficient')
ax.legend()
ax.set_xlabel("Weibull parameter $a$")
ax.set_ylabel("Gini coefficient")
plt.show()
```

The simulation shows that the fit is good.

31.4 A Model of Wealth Dynamics

Having discussed inequality measures, let us now turn to wealth dynamics.

The model we will study is

$$w_{t+1} = (1 + r_{t+1})s(w_t) + y_{t+1} \tag{1}$$

where

- w_t is wealth at time t for a given household,
- r_t is the rate of return of financial assets,
- y_t is current non-financial (e.g., labor) income and
- $s(w_t)$ is current wealth net of consumption

Letting $\{z_t\}$ be a correlated state process of the form

$$z_{t+1} = az_t + b + \sigma_z \epsilon_{t+1}$$

we'll assume that

$$R_t := 1 + r_t = c_r \exp(z_t) + \exp(\mu_r + \sigma_r \xi_t)$$

and

$$y_t = c_y \exp(z_t) + \exp(\mu_y + \sigma_y \zeta_t)$$

Here $\{(\epsilon_t, \xi_t, \zeta_t)\}$ is IID and standard normal in $\mathbb{R}^3$.

The value of c_r should be close to zero, since rates of return on assets do not exhibit large trends.

When we simulate a population of households, we will assume all shocks are idiosyncratic (i.e., specific to individual households and independent across them).

Regarding the savings function s, our default model will be

$$s(w) = s_0 w \cdot \mathbb{1}\{w \geq \hat{w}\} \tag{2}$$

where s_0 is a positive constant.

Thus, for $w < \hat{w}$, the household saves nothing. For $w \geq \bar{w}$, the household saves a fraction s_0 of their wealth.

We are using something akin to a fixed savings rate model, while acknowledging that low wealth households tend to save very little.

31.5 Implementation

Here's some type information to help Numba.

```
In [7]: wealth_dynamics_data = [
            ('w_hat',  float64),    # savings parameter
            ('s_0',    float64),    # savings parameter
            ('c_y',    float64),    # labor income parameter
            ('μ_y',    float64),    # labor income paraemter
            ('σ_y',    float64),    # labor income parameter
            ('c_r',    float64),    # rate of return parameter
            ('μ_r',    float64),    # rate of return parameter
            ('σ_r',    float64),    # rate of return parameter
            ('a',      float64),    # aggregate shock parameter
            ('b',      float64),    # aggregate shock parameter
            ('σ_z',    float64),    # aggregate shock parameter
            ('z_mean', float64),    # mean of z process
            ('z_var', float64),     # variance of z process
            ('y_mean', float64),    # mean of y process
            ('R_mean', float64)     # mean of R process
        ]
```

Here's a class that stores instance data and implements methods that update the aggregate state and household wealth.

In [8]:
```
@jitclass(wealth_dynamics_data)
class WealthDynamics:

    def __init__(self,
                 w_hat=1.0,
                 s_0=0.75,
                 c_y=1.0,
                 μ_y=1.0,
                 σ_y=0.2,
                 c_r=0.05,
                 μ_r=0.1,
                 σ_r=0.5,
                 a=0.5,
                 b=0.0,
                 σ_z=0.1):

        self.w_hat, self.s_0 = w_hat, s_0
        self.c_y, self.μ_y, self.σ_y = c_y, μ_y, σ_y
        self.c_r, self.μ_r, self.σ_r = c_r, μ_r, σ_r
        self.a, self.b, self.σ_z = a, b, σ_z

        # Record stationary moments
        self.z_mean = b / (1 - a)
        self.z_var = σ_z**2 / (1 - a**2)
        exp_z_mean = np.exp(self.z_mean + self.z_var / 2)
        self.R_mean = c_r * exp_z_mean + np.exp(μ_r + σ_r**2 / 2)
        self.y_mean = c_y * exp_z_mean + np.exp(μ_y + σ_y**2 / 2)

        # Test a stability condition that ensures wealth does not diverge
        # to infinity.
        α = self.R_mean * self.s_0
        if α >= 1:
            raise ValueError("Stability condition failed.")

    def parameters(self):
        """
        Collect and return parameters.
        """
        parameters = (self.w_hat, self.s_0,
                      self.c_y, self.μ_y, self.σ_y,
                      self.c_r, self.μ_r, self.σ_r,
                      self.a, self.b, self.σ_z)
        return parameters

    def update_states(self, w, z):
        """
        Update one period, given current wealth w and persistent
        state z.
        """

        # Simplify names
        params = self.parameters()
        w_hat, s_0, c_y, μ_y, σ_y, c_r, μ_r, σ_r, a, b, σ_z = params
        zp = a * z + b + σ_z * np.random.randn()

        # Update wealth
        y = c_y * np.exp(zp) + np.exp(μ_y + σ_y * np.random.randn())
        wp = y
```

```
        if w >= w_hat:
            R = c_r * np.exp(zp) + np.exp(μ_r + σ_r * np.random.randn())
            wp += R * s_0 * w
        return wp, zp
```

Here's function to simulate the time series of wealth for in individual households.

```
In [9]: @njit
        def wealth_time_series(wdy, w_0, n):
            """
            Generate a single time series of length n for wealth given
            initial value w_0.

            The initial persistent state z_0 for each household is drawn from
            the stationary distribution of the AR(1) process.

                * wdy: an instance of WealthDynamics
                * w_0: scalar
                * n: int


            """
            z = wdy.z_mean + np.sqrt(wdy.z_var) * np.random.randn()
            w = np.empty(n)
            w[0] = w_0
            for t in range(n-1):
                w[t+1], z = wdy.update_states(w[t], z)
            return w
```

Now here's function to simulate a cross section of households forward in time.

Note the use of parallelization to speed up computation.

```
In [10]: @njit(parallel=True)
         def update_cross_section(wdy, w_distribution, shift_length=500):
             """
             Shifts a cross-section of household forward in time

             * wdy: an instance of WealthDynamics
             * w_distribution: array_like, represents current cross-section

             Takes a current distribution of wealth values as w_distribution
             and updates each w_t in w_distribution to w_{t+j}, where
             j = shift_length.

             Returns the new distribution.

             """
             new_distribution = np.empty_like(w_distribution)

             # Update each household
             for i in prange(len(new_distribution)):
                 z = wdy.z_mean + np.sqrt(wdy.z_var) * np.random.randn()
                 w = w_distribution[i]
                 for t in range(shift_length-1):
                     w, z = wdy.update_states(w, z)
```

```
        new_distribution[i] = w
    return new_distribution
```

Parallelization is very effective in the function above because the time path of each household can be calculated independently once the path for the aggregate state is known.

31.6 Applications

Let's try simulating the model at different parameter values and investigate the implications for the wealth distribution.

31.6.1 Time Series

Let's look at the wealth dynamics of an individual household.

```
In [11]: wdy = WealthDynamics()

         ts_length = 200
         w = wealth_time_series(wdy, wdy.y_mean, ts_length)

         fig, ax = plt.subplots()
         ax.plot(w)
         plt.show()
```

Notice the large spikes in wealth over time.

Such spikes are similar to what we observed in time series when we studied Kesten processes.

31.6.2 Inequality Measures

Let's look at how inequality varies with returns on financial assets.

The next function generates a cross section and then computes the Lorenz curve and Gini coefficient.

```
In [12]: def generate_lorenz_and_gini(wdy, num_households=100_000, T=500):
             """
             Generate the Lorenz curve data and gini coefficient corresponding to a
             WealthDynamics mode by simulating num_households forward to time T.
             """
             ψ_0 = np.ones(num_households) * wdy.y_mean
             z_0 = wdy.z_mean

             ψ_star = update_cross_section(wdy, ψ_0, shift_length=T)
             return qe.gini_coefficient(ψ_star), qe.lorenz_curve(ψ_star)
```

Now we investigate how the Lorenz curves associated with the wealth distribution change as return to savings varies.

The code below plots Lorenz curves for three different values of μ_r.

If you are running this yourself, note that it will take one or two minutes to execute.

This is unavoidable because we are executing a CPU intensive task.

In fact the code, which is JIT compiled and parallelized, runs extremely fast relative to the number of computations.

```
In [13]: fig, ax = plt.subplots()
         μ_r_vals = (0.0, 0.025, 0.05)
         gini_vals = []

         for μ_r in μ_r_vals:
             wdy = WealthDynamics(μ_r=μ_r)
             gv, (f_vals, l_vals) = generate_lorenz_and_gini(wdy)
             ax.plot(f_vals, l_vals, label=f'$\psi^*$ at $\mu_r = {μ_r:0.2}$')
             gini_vals.append(gv)

         ax.plot(f_vals, f_vals, label='equality')
         ax.legend(loc="upper left")
         plt.show()
```

The Lorenz curve shifts downwards as returns on financial income rise, indicating a rise in inequality.

We will look at this again via the Gini coefficient immediately below, but first consider the following image of our system resources when the code above is executing:

Notice how effectively Numba has implemented multithreading for this routine: all 8 CPUs on our workstation are running at maximum capacity (even though four of them are virtual).

Since the code is both efficiently JIT compiled and fully parallelized, it's close to impossible to make this sequence of tasks run faster without changing hardware.

Now let's check the Gini coefficient.

In [14]:
```
fig, ax = plt.subplots()
ax.plot(μ_r_vals, gini_vals, label='gini coefficient')
ax.set_xlabel("$\mu_r$")
ax.legend()
plt.show()
```

Once again, we see that inequality increases as returns on financial income rise.

Let's finish this section by investigating what happens when we change the volatility term σ_r in financial returns.

In [15]:
```
fig, ax = plt.subplots()
σ_r_vals = (0.35, 0.45, 0.52)
gini_vals = []

for σ_r in σ_r_vals:
    wdy = WealthDynamics(σ_r=σ_r)
    gv, (f_vals, l_vals) = generate_lorenz_and_gini(wdy)
    ax.plot(f_vals, l_vals, label=f'$\psi^*$ at $\sigma_r = {σ_r:0.2}$')
    gini_vals.append(gv)

ax.plot(f_vals, f_vals, label='equality')
ax.legend(loc="upper left")
plt.show()
```

We see that greater volatility has the effect of increasing inequality in this model.

31.7 Exercises

31.7.1 Exercise 1

For a wealth or income distribution with Pareto tail, a higher tail index suggests lower inequality.

Indeed, it is possible to prove that the Gini coefficient of the Pareto distribution with tail index a is $1/(2a-1)$.

To the extent that you can, confirm this by simulation.

In particular, generate a plot of the Gini coefficient against the tail index using both the theoretical value just given and the value computed from a sample via `qe.gini_coefficient`.

For the values of the tail index, use `a_vals = np.linspace(1, 10, 25)`.

Use sample of size 1,000 for each a and the sampling method for generating Pareto draws employed in the discussion of Lorenz curves for the Pareto distribution.

To the extend that you can, interpret the monotone relationship between the Gini index and a.

31.7.2 Exercise 2

The wealth process (1) is similar to a Kesten process.

This is because, according to (2), savings is constant for all wealth levels above $\hat{w}$.

When savings is constant, the wealth process has the same quasi-linear structure as a Kesten process, with multiplicative and additive shocks.

The Kesten–Goldie theorem tells us that Kesten processes have Pareto tails under a range of parameterizations.

The theorem does not directly apply here, since savings is not always constant and since the multiplicative and additive terms in (1) are not IID.

At the same time, given the similarities, perhaps Pareto tails will arise.

To test this, run a simulation that generates a cross-section of wealth and generate a rank-size plot.

In viewing the plot, remember that Pareto tails generate a straight line. Is this what you see?

For sample size and initial conditions, use

```
In [16]: num_households = 250_000
         T = 500                                        # shift forward T periods
         ψ_0 = np.ones(num_households) * wdy.y_mean     # initial distribution
         z_0 = wdy.z_mean
```

31.8 Solutions

Here is one solution, which produces a good match between theory and simulation.

31.8.1 Exercise 1

```
In [17]: a_vals = np.linspace(1, 10, 25)  # Pareto tail index
         ginis = np.empty_like(a_vals)

         n = 1000                         # size of each sample
         fig, ax = plt.subplots()
         for i, a in enumerate(a_vals):
             y = np.random.uniform(size=n)**(-1/a)
             ginis[i] = qe.gini_coefficient(y)
         ax.plot(a_vals, ginis, label='sampled')
         ax.plot(a_vals, 1/(2*a_vals - 1), label='theoretical')
         ax.legend()
         plt.show()
```

In general, for a Pareto distribution, a higher tail index implies less weight in the right hand tail.

This means less extreme values for wealth and hence more equality.

More equality translates to a lower Gini index.

31.8.2 Exercise 2

First let's generate the distribution:

```
In [18]: num_households = 250_000
         T = 500  # how far to shift forward in time
         ψ_0 = np.ones(num_households) * wdy.y_mean
         z_0 = wdy.z_mean

         ψ_star = update_cross_section(wdy, ψ_0, shift_length=T)
```

Here's a function to produce rank-size data for the plot.

```
In [19]: def rank_size_data(data, c=0.001):
             """
             Generate rank-size data corresponding to distribution data.

                 * data is array like
                 * c is a float indicating the top (c x 100)% of the
                   distribution
             """
             w = - np.sort(- data)                        # Reverse sort
             w = w[:int(len(w) * c)]                      # extract top c%
             rank_data = np.log(np.arange(len(w)) + 1)
             size_data = np.log(w)
             return rank_data, size_data
```

Now let's see the rank-size plot.

```
In [20]: rank_data, size_data = rank_size_data(ψ_star)

         fig, ax = plt.subplots()
         ax.plot(rank_data, size_data, 'o', markersize=3.0, alpha=0.5)

         ax.set_xlabel("log rank")
         ax.set_ylabel("log size")

         plt.show()
```

Chapter 32

Continuous State Markov Chains

32.1 Contents

- Overview 32.2
- The Density Case 32.3
- Beyond Densities 32.4
- Stability 32.5
- Exercises 32.6
- Solutions 32.7
- Appendix 32.8

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

```
In [1]: !pip install --upgrade quantecon
```

32.2 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., [117]) in using the term to refer to any discrete **time** Markov process.

Let's begin with some imports:

In [2]:
```
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
```

32.3 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.

32.3.1 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

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

What kind of model does p_w represent?

The answer is, the (normally distributed) random walk

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

To see this, let's find the stochastic kernel p corresponding to (2).

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

Letting $X_t = x$ in (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 (1).

32.3.2 Connection to Stochastic Difference Equations

In the previous section, we made the connection between stochastic difference equation (2) and stochastic kernel (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

$$X_{t+1} = \mu(X_t) + \sigma(X_t)\,\xi_{t+1} \tag{3}$$

Here we assume that

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

Example 1: The random walk (2) is a special case of (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

$$X_{t+1} = \alpha X_t + (\beta + \gamma X_t^2)^{1/2}\xi_{t+1} \tag{4}$$

This is a special case of (3) with $\mu(x) = \alpha x$ and $\sigma(x) = (\beta + \gamma x^2)^{1/2}$.

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

$$k_{t+1} = sA_{t+1}f(k_t) + (1-\delta)k_t \tag{5}$$

Here

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

(The fixed savings rate can be rationalized as the optimal policy for a particular set of technologies and preferences (see [108], section 3.1.2), although we omit the details here).

Equation (5) is a special case of (3) with $\mu(x) = (1-\delta)x$ and $\sigma(x) = sf(x)$.

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

$$f_V(v) = \frac{1}{b} f_U\left(\frac{v-a}{b}\right) \tag{6}$$

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

Taking (6) as given for the moment, we can obtain the stochastic kernel p for (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 (6),

$$p(x, y) = \frac{1}{\sigma(x)} \phi \left(\frac{y - \mu(x)}{\sigma(x)} \right) \tag{7}$$

For example, the growth model in (5) has stochastic kernel

$$p(x, y) = \frac{1}{sf(x)} \phi \left(\frac{y - (1 - \delta)x}{sf(x)} \right) \tag{8}$$

where ϕ is the density of A_{t+1}.

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

32.3.3 Distribution Dynamics

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

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

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

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

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

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

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

$$\psi_{t+1}(y) = \int p(x, y) \psi_t(x) \, dx, \qquad \forall y \in S \tag{9}$$

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

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

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

$$(\psi P)(y) = \int p(x, y) \psi(x) dx \tag{10}$$

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

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

With this notation, we can write (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,

$$\psi_{t+1} = \psi_t P \tag{11}$$

Equation (11) tells us that if we specify a distribution for ψ_0, then the entire sequence of future distributions can be obtained by iterating with P.

It's interesting to note that (11) is a deterministic difference equation.

Thus, by converting a stochastic difference equation such as (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.

32.3.4 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 (10) and (11) using numerical integration.

However, to produce ψP from ψ via (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:

$$k_{t+1} = sA_{t+1}f(k_t) + (1-\delta)k_t \tag{12}$$

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

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

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

1. draw k_0 from the specified initial condition ψ_0
2. draw the shocks $A_1, \dots, A_t$ from their specified density ϕ
3. compute k_t iteratively via (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 ψ_t.

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

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

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

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

$$\psi_t^n(y) = \frac{1}{n} \sum_{i=1}^n p(k_{t-1}^i, y) \tag{13}$$

where p is the growth model stochastic kernel in (8).

What is the justification for this slightly surprising estimator?

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

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

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

Here the first equality is by the definition of ψ_{t-1}, and the second is by (9).

We have just shown that our estimator $\psi_t^n(y)$ in (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).

32.3.5 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 (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.

32.3.6 Example

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

In [3]:
```
# == Define parameters == #
s = 0.2
δ = 0.1
a_σ = 0.4                    # A = exp(B) where B ~ N(0, a_σ)
α = 0.4                      # We set f(k) = k**α
ψ_0 = beta(5, 5, scale=0.5)  # Initial distribution
ϕ = lognorm(a_σ)


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

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

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

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

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

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

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

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

32.4 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.

32.4.1 Example and Definitions

To illustrate the issues, recall that Hopenhayn and Rogerson [87] 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 (9) are then replaced by

$$F_{t+1}(y) = \int G(x, y) F_t(dx) \tag{14}$$

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

The intuition behind (14) is essentially the same as for (9).

32.4.2 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.

32.5 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.

32.5.1 Theoretical Results

Analogous to the finite case, given a stochastic kernel p and corresponding Markov operator as defined in (10), a density ψ^* on S is called *stationary* for P if it is a fixed point of the operator P.

In other words,

$$\psi^*(y) = \int p(x, y)\psi^*(x)\, dx, \qquad \forall y \in S \tag{15}$$

As with the finite case, if ψ^* is stationary for P, and the distribution of X_0 is ψ^*, then, in view of (11), X_t will have this same distribution for all t.

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

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

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

Even existence can fail very easily.

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

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

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

$$\forall\, \psi \in \mathcal{D}, \quad \psi P^t \to \psi^* \quad \text{as} \quad t \to \infty \tag{16}$$

This combination of existence, uniqueness and global convergence in the sense of (16) is often referred to as *global stability*.

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

$$\frac{1}{n}\sum_{t=1}^{n} h(X_t) \to \int h(x)\psi^*(x)dx \quad \text{as } n \to \infty \tag{17}$$

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

Note that the convergence in (17) does not depend on the distribution (or value) of X_0.

This is actually very important for simulation — it means we can learn about ψ^* (i.e., approximate the right-hand side of (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

- [149] contains a classic (but slightly outdated) treatment of these topics.
- From the mathematical literature, [103] and [117] 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.

32.5.2 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 (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.

32.5.3 Computing Stationary Densities

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

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

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

Let's say that we have a model of the form (3) that is stable and ergodic.

Let p be the corresponding stochastic kernel, as given in (7).

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

$$\psi_n^*(y) = \frac{1}{n} \sum_{t=1}^{n} p(X_t, y) \tag{18}$$

This is essentially the same as the look-ahead estimator (13), except that now the observations we generate are a single time-series, rather than a cross-section.

The justification for (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 (17) and the equality on the right is by (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.

32.6 Exercises

32.6.1 Exercise 1

Consider the simple threshold autoregressive model

$$X_{t+1} = \theta |X_t| + (1 - \theta^2)^{1/2} \xi_{t+1} \qquad \text{where} \quad \{\xi_t\} \overset{\text{IID}}{\sim} N(0, 1) \tag{19}$$

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

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

$$\psi^*(y) = 2 \, \phi(y) \, \Phi \left[\frac{\theta y}{(1 - \theta^2)^{1/2}} \right] \tag{20}$$

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

As an exercise, compute the look-ahead estimate of ψ^*, as defined in (18), and compare it with ψ^* in (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.

32.6.2 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)
```

32.6.3 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:

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

Here is the code and figure:

```
In [4]: 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 (19).

We know that the distribution of X_t will converge to (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 (19).
2. Create a boxplot representing n distributions, where the t-th distribution shows the k observations of X_t.

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

32.7 Solutions

32.7.1 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

In [5]:
```
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()
```

32.7.2 Exercise 2

Here's one program that does the job

```
In [6]: # == 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)
            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()
```

32.7.3 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

```
In [7]: 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

32.8 Appendix

Here's the proof of (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 (6).

Chapter 33

Cass-Koopmans Optimal Growth Model

33.1 Contents

- Overview 33.2
- The Growth Model 33.3
- Competitive Equilibrium 33.4

Coauthor: Brandon Kaplowitz

33.2 Overview

This lecture describes a model that Tjalling Koopmans [99] and David Cass [33] used to analyze optimal growth.

The model can be viewed as an extension of the model of Robert Solow described in an earlier lecture but adapted to make the savings rate the outcome of an optimal choice.

(Solow assumed a constant saving rate determined outside the model).

We describe two versions of the model to illustrate what is, in fact, a more general connection between a **planned economy** and an economy organized as a **competitive equilibrium**.

The lecture uses important ideas including

- Hicks-Arrow prices named after John R. Hicks and Kenneth Arrow.
- A min-max problem for solving a planning problem.
- A **shooting algorithm** for solving difference equations subject to initial and terminal conditions.
- A connection between some Lagrange multipliers in the min-max problem and the Hicks-Arrow prices.
- A **Big** K **, little** k trick widely used in macroeconomic dynamics.

    - We shall encounter this trick in this lecture and also in this lecture.

- An application of a **guess and verify** method for solving a system of difference equations.

- The intimate connection between the cases for the optimality of two competing visions of good ways to organize an economy, namely:

    - **socialism** in which a central planner commands the allocation of resources, and

- **capitalism** (also known as **a free markets economy**) in which competitive equilibrium **prices** induce individual consumers and producers to choose a socially optimal allocation as an unintended consequence of their completely selfish decisions
- A **turnpike** property that describes optimal paths for long-but-finite horizon economies.
- A non-stochastic version of a theory of the **term structure of interest rates**.

Let's start with some standard imports:

```
In [1]: from numba import njit
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

33.3 The Growth Model

Time is discrete and takes values $t = 0, 1, \ldots, T$.

(We leave open the possibility that $T = +\infty$, but that will require special care in interpreting and using a **terminal condition** on K_t at $t = T + 1$ to be described below).

A single good can either be consumed or invested in physical capital.

The consumption good is not durable and depreciates completely if not consumed immediately.

The capital good is durable but depreciates each period at rate $\delta \in (0, 1)$.

We let C_t be a nondurable consumption good at time t.

Let K_t be the stock of physical capital at time t.

Let $\vec{C} = \{C_0, \ldots, C_T\}$ and $\vec{K} = \{K_1, \ldots, K_{T+1}\}$.

A representative household is endowed with one unit of labor at each t and likes the consumption good at each t.

The representative household inelastically supplies a single unit of labor N_t at each t, so that $N_t = 1$ for all $t \in [0, T]$.

The representative household has preferences over consumption bundles ordered by the utility functional:

$$U(\vec{C}) = \sum_{t=0}^{T} \beta^t \frac{C_t^{1-\gamma}}{1-\gamma} \tag{1}$$

where $\beta \in (0, 1)$ is a discount factor and $\gamma > 0$ governs the curvature of the one-period utility function.

Note that

$$u(C_t) = \frac{C_t^{1-\gamma}}{1-\gamma} \tag{2}$$

satisfies $u' > 0, u'' < 0$.

$u' > 0$ asserts the consumer prefers more to less.

$u'' < 0$ asserts that marginal utility declines with increases in C_t.

We assume that $K_0 > 0$ is a given exogenous level of initial capital.

There is an economy-wide production function

$$F(K_t, N_t) = AK_t^{\alpha} N_t^{1-\alpha} \tag{3}$$

with $0 < \alpha < 1$, $A > 0$.

A feasible allocation $\vec{C}, \vec{K}$ satisfies

$$C_t + K_{t+1} \leq F(K_t, N_t) + (1-\delta)K_t, \quad \text{for all } t \in [0, T] \tag{4}$$

where $\delta \in (0, 1)$ is a depreciation rate of capital.

33.3.1 Planning Problem

A planner chooses an allocation $\{\vec{C}, \vec{K}\}$ to maximize (1) subject to (4).

Let $\vec{\mu} = \{\mu_0, \dots, \mu_T\}$ be a sequence of nonnegative **Lagrange multipliers**.

To find an optimal allocation, we form a Lagrangian

$$\mathcal{L}(\vec{C}, \vec{K}, \vec{\mu}) = \sum_{t=0}^{T} \beta^t \left\{ u(C_t) + \mu_t \left(F(K_t, 1) + (1-\delta)K_t - C_t - K_{t+1} \right) \right\}$$

and then solve the following min-max problem:

$$\min_{\vec{\mu}} \max_{\vec{C}, \vec{K}} \mathcal{L}(\vec{C}, \vec{K}, \vec{\mu}) \tag{5}$$

Useful Properties of Linearly Homogeneous Production Function

The following technicalities will help us.

Notice that

$$F(K_t, N_t) = AK_t^{\alpha} N_t^{1-\alpha} = N_t A \left(\frac{K_t}{N_t} \right)^{\alpha}$$

Define the **output per-capita production function**

$$f\left(\frac{K_t}{N_t} \right) = A \left(\frac{K_t}{N_t} \right)^{\alpha}$$

whose argument is **capital per-capita**.

Evidently,

$$F(K_t, N_t) = N_t f\left(\frac{K_t}{N_t}\right)$$

Now for some useful calculations.

First

$$\begin{aligned}
\frac{\partial F}{\partial K} &= \frac{\partial N_t f\left(\frac{K_t}{N_t}\right)}{\partial K_t} \\
&= N_t f'\left(\frac{K_t}{N_t}\right)\frac{1}{N_t} \quad \text{(Chain rule)} \\
&= f'\left(\frac{K_t}{N_t}\right)\Big|_{N_t=1} \\
&= f'(K_t)
\end{aligned} \tag{6}$$

Also

$$\begin{aligned}
\frac{\partial F}{\partial N} &= \frac{\partial N_t f\left(\frac{K_t}{N_t}\right)}{\partial N_t} \quad \text{(Product rule)} \\
&= f\left(\frac{K_t}{N_t}\right) + N_t f'\left(\frac{K_t}{N_t}\right)\frac{-K_t}{N_t^2} \quad \text{(Chain rule)} \\
&= f\left(\frac{K_t}{N_t}\right) - \frac{K_t}{N_t} f'\left(\frac{K_t}{N_t}\right)\Big|_{N_t=1} \\
&= f(K_t) - f'(K_t)K_t
\end{aligned}$$

Back to Solving the Problem

To solve the Lagrangian extremization problem, we compute first derivatives of the Lagrangian and set them equal to 0.

- **Note:** Our objective function and constraints satisfy conditions that work to assure that required second-order conditions are satisfied at an allocation that satisfies the first-order conditions that we are about to compute.

Here are the **first order necessary conditions** for extremization (i.e., maximization with respect to $\vec{C}, \vec{K}$, minimization with respect to $\vec{\mu}$):

$$C_t: \qquad u'(C_t) - \mu_t = 0 \qquad \text{for all} \quad t = 0, 1, \dots, T \tag{7}$$

$$K_t: \qquad \beta\mu_t\left[(1-\delta) + f'(K_t)\right] - \mu_{t-1} = 0 \qquad \text{for all} \quad t = 1, 2, \dots, T \tag{8}$$

$$\mu_t: \qquad F(K_t, 1) + (1-\delta)K_t - C_t - K_{t+1} = 0 \qquad \text{for all} \quad t = 0, 1, \dots, T \tag{9}$$

$$K_{T+1}: \qquad -\mu_T \leq 0, \ \leq 0 \text{ if } K_{T+1} = 0; \ = 0 \text{ if } K_{T+1} > 0 \tag{10}$$

Note that in (8) we plugged in for $\frac{\partial F}{\partial K}$ using our formula (6) above.

Because $N_t = 1$ for $t = 1, \dots, T$, need not differentiate with respect to those arguments.

Note that (9) comes from the occurrence of K_t in both the period t and period $t-1$ feasibility constraints.

(10) comes from differentiating with respect to K_{T+1} in the last period and applying the following condition called a **Karush-Kuhn-Tucker condition** (KKT):

$$\mu_T K_{T+1} = 0 \tag{11}$$

See Karush-Kuhn-Tucker conditions.

Combining (7) and (8) gives

$$u'(C_t)\left[(1-\delta) + f'(K_t)\right] - u'(C_{t-1}) = 0 \quad \text{for all } t = 1, 2, \dots, T+1$$

Rewriting gives

$$u'(C_{t+1})\left[(1-\delta) + f'(K_{t+1})\right] = u'(C_t) \quad \text{for all } t = 0, 1, \dots, T \tag{12}$$

Taking the inverse of the utility function on both sides of the above equation gives

$$C_{t+1} = u'^{-1}\left(\left(\frac{\beta}{u'(C_t)}[f'(K_{t+1}) + (1-\delta)]\right)^{-1}\right)$$

or using our utility function (2)

$$\begin{aligned} C_{t+1} &= \left(\beta C_t^{\gamma}[f'(K_{t+1}) + (1-\delta)]\right)^{1/\gamma} \\ &= C_t\left(\beta[f'(K_{t+1}) + (1-\delta)]\right)^{1/\gamma} \end{aligned}$$

The above first-order condition for consumption is called an **Euler equation**.

It tells us how consumption in adjacent periods are optimally related to each other and to capital next period.

We now use some of the equations above to calculate some variables and functions that we'll soon use to solve the planning problem with Python.

```
In [2]: @njit
        def u(c, γ):
            '''
            Utility function
            ASIDE: If you have a utility function that is hard to solve by hand
            you can use automatic or symbolic  differentiation
            See https://github.com/HIPS/autograd
            '''
            if γ == 1:
                # If γ = 1 we can show via L'hopital's Rule that the utility
```

```
        # becomes log
        return np.log(c)
    else:
        return c**(1 - γ) / (1 - γ)

@njit
def u_prime(c, γ):
    '''Derivative of utility'''
    if γ == 1:
        return 1 / c
    else:
        return c**(-γ)

@njit
def u_prime_inv(c, γ):
    '''Inverse utility'''
    if γ == 1:
        return c
    else:
        return c**(-1 / γ)

@njit
def f(A, k, α):
    '''Production function'''
    return A * k**α

@njit
def f_prime(A, k, α):
    '''Derivative of production function'''
    return α * A * k**(α - 1)

@njit
def f_prime_inv(A, k, α):
    return (k / (A * α))**(1 / (α - 1))
```

33.3.2 Shooting Method

We shall use a **shooting method** to compute an optimal allocation $\vec{C}, \vec{K}$ and an associated Lagrange multiplier sequence $\vec{\mu}$.

The first-order necessary conditions for the planning problem, namely, equations (7), (8), and (9), form a system of **difference equations** with two boundary conditions:

- K_0 is a given **initial condition** for capital
- $K_{T+1} = 0$ is a **terminal condition** for capital that we deduced from the first-order necessary condition for K_{T+1} the KKT condition (11)

We have no initial condition for the Lagrange multiplier μ_0.

If we did, solving for the allocation would be simple:

- Given μ_0 and k_0, we could compute c_0 from equation (7) and then k_1 from equation (9) and μ_1 from equation (8).
- We could then iterate on to compute the remaining elements of $\vec{C}, \vec{K}, \vec{\mu}$.

But we don't have an initial condition for μ_0, so this won't work.

But a simple modification called the **shooting algorithm** will work.

The **shooting algorithm** is an instance of a **guess and verify** algorithm.

It proceeds as follows:

- Guess a value for the initial Lagrange multiplier μ_0.
- Apply the **simple algorithm** described above.
- Compute the implied value of k_{T+1} and check whether it equals zero.
- If the implied $K_{T+1} = 0$, we have solved the problem.
- If $K_{T+1} > 0$, lower μ_0 and try again.
- If $K_{T+1} < 0$, raise μ_0 and try again.

The following Python code implements the shooting algorithm for the planning problem.

We make a slight modification starting with a guess of c_0 but since c_0 is a function of μ_0 there is no difference to the procedure above.

We'll apply it with an initial guess that will turn out not to be perfect, as we'll soon see.

In [3]:
```
# Parameters
γ = 2
δ = 0.02
β = 0.95
α = 0.33
A = 1

# Initial guesses
T = 10
c = np.zeros(T+1)  # T periods of consumption initialized to 0
# T periods of capital initialized to 0 (T+2 to include t+1 variable as well)
k = np.zeros(T+2)
k[0] = 0.3  # Initial k
c[0] = 0.2  # Guess of c_0

@njit
def shooting_method(c, # Initial consumption
                    k,   # Initial capital
                    γ,   # Coefficient of relative risk aversion
                    δ,   # Depreciation rate on capital# Depreciation rate
                    β,   # Discount factor
                    α,   # Return to capital per capita
                    A):  # Technology

    T = len(c) - 1

    for t in range(T):
        # Equation 1 with inequality
        k[t+1] = f(A=A, k=k[t], α=α) + (1 - δ) * k[t] - c[t]
        if k[t+1] < 0:   # Ensure nonnegativity
            k[t+1] = 0

      # Equation 2: We keep in the general form to show how we would
      # solve if we didn't want to do any simplification

        if β * (f_prime(A=A, k=k[t+1], α=α) + (1 - δ)) == np.inf:
            # This only occurs if k[t+1] is 0, in which case, we won't
           # produce anything next period, so consumption will have to be 0
            c[t+1] = 0
        else:
            c[t+1] = u_prime_inv(u_prime(c=c[t], γ=γ) \
```

```
            / (β * (f_prime(A=A, k=k[t+1], α=α) + (1 - δ))), γ=γ)

    # Terminal condition calculation
    k[T+1] = f(A=A, k=k[T], α=α) + (1 - δ) * k[T] - c[T]

    return c, k

paths = shooting_method(c, k, γ, δ, β, α, A)

fig, axes = plt.subplots(1, 2, figsize=(10, 4))
colors = ['blue', 'red']
titles = ['Consumption', 'Capital']
ylabels = ['$c_t$', '$k_t$']

for path, color, title, y, ax in zip(paths, colors, titles, ylabels, axes):
    ax.plot(path, c=color, alpha=0.7)
    ax.set(title=title, ylabel=y, xlabel='t')

ax.scatter(T+1, 0, s=80)
ax.axvline(T+1, color='k', ls='--', lw=1)

plt.tight_layout()
plt.show()
```

Evidently, our initial guess for μ_0 is too high and makes initial consumption is too low.

We know this because we miss our $K_{T+1} = 0$ target on the high side.

Now we automate things with a search-for-a-good μ_0 algorithm that stops when we hit the target $K_{t+1} = 0$.

The search procedure is to use a **bisection method**.

Here is how we apply the bisection method.

We take an initial guess for C_0 (we can eliminate μ_0 because C_0 is an exact function of μ_0).

We know that the lowest C_0 can ever be is 0 and the largest it can be is initial output $f(K_0)$.

We take a C_0 guess and shoot forward to $T + 1$.

If the $K_{T+1} > 0$, let it be our new **lower** bound on C_0.

If $K_{T+1} < 0$, let it be our new **upper** bound.

Make a new guess for C_0 exactly halfway between our new upper and lower bounds.

Shoot forward again and iterate the procedure.

When K_{T+1} gets close enough to 0 (within some error tolerance bounds), stop and declare victory.

```
In [4]: @njit
        def bisection_method(c,
                             k,
                            γ,              # Coefficient of relative risk aversion
                             δ,              # Depreciation rate
                             β,              # Discount factor
                             α,              # Return to capital per capita
                             A,              # Technology
                             tol=1e-4,
                             max_iter=1e4,
                             terminal=0):    # Value we are shooting towards

            T = len(c) - 1
            i = 1                            # Initial iteration
            c_high = f(k=k[0], α=α, A=A)     # Initial high value of c
            c_low = 0                        # Initial low value of c

            path_c, path_k = shooting_method(c, k, γ, δ, β, α, A)

           while (np.abs((path_k[T+1] - terminal)) > tol or path_k[T] == terminal) \
                and i < max_iter:

                if path_k[T+1] - terminal > tol:
                    # If assets are too high the c[0] we chose is now a lower bound
                    # on possible values of c[0]
                    c_low = c[0]
                elif path_k[T+1] - terminal < -tol:
                    # If assets fell too quickly, the c[0] we chose is now an upper
                    # bound on possible values of c[0]
                    c_high=c[0]
                elif path_k[T] == terminal:
                   # If assets fell  too quickly, the c[0] we chose is now an upper
                    # bound on possible values of c[0]
                    c_high=c[0]

                c[0] = (c_high + c_low) / 2  # This is the bisection part
                path_c, path_k = shooting_method(c, k, γ, δ, β, α, A)
                i += 1

            if np.abs(path_k[T+1] - terminal) < tol and path_k[T] != terminal:
                print('Converged successfully on iteration', i-1)
            else:
                print('Failed to converge and hit maximum iteration')

            μ = u_prime(c=path_c, γ=γ)
            return path_c, path_k, μ
```

Now we can plot

```
In [5]: T = 10
        c = np.zeros(T+1) # T periods of consumption initialized to 0
```

```
# T periods of capital initialized to 0. T+2 to include t+1 variable as well
k = np.zeros(T+2)

k[0] = 0.3 # initial k
c[0] = 0.3 # our guess of c_0

paths = bisection_method(c, k, γ, δ, β, α, A)

def plot_paths(paths, axes=None, ss=None):

    T = len(paths[0])

    if axes is None:
        fix, axes = plt.subplots(1, 3, figsize=(13, 3))

    ylabels = ['$c_t$', '$k_t$', '$\mu_t$']
    titles = ['Consumption', 'Capital', 'Lagrange Multiplier']

    for path, y, title, ax in zip(paths, ylabels, titles, axes):
        ax.plot(path)
        ax.set(ylabel=y, title=title, xlabel='t')

    # Plot steady state value of capital
    if ss is not None:
        axes[1].axhline(ss, c='k', ls='--', lw=1)

    axes[1].axvline(T, c='k', ls='--', lw=1)
    axes[1].scatter(T, paths[1][-1], s=80)
    plt.tight_layout()

plot_paths(paths)
```

```
Converged successfully on iteration 18
```

33.3.3 Setting Intial Capital to the Steady State

If $T \to +\infty$, the optimal allocation converges to steady state values of C_t and K_t.

It is instructive to compute these and then to set K_0 equal to its steady state value.

In a steady state $K_{t+1} = K_t = \bar{K}$ for all very large t the feasibility constraint (4) is

$$f(\bar{K}) - \delta\bar{K} = \bar{C} \tag{13}$$

Substituting $K_t = \bar{K}$ and $C_t = \bar{C}$ for all t into (12) gives

$$1 = \beta \frac{u'(\bar{C})}{u'(\bar{C})}[f'(\bar{K}) + (1-\delta)]$$

Defining $\beta = \frac{1}{1+\rho}$, and cancelling gives

$$1 + \rho = 1[f'(\bar{K}) + (1-\delta)]$$

Simplifying gives

$$f'(\bar{K}) = \rho + \delta$$

and

$$\bar{K} = f'^{-1}(\rho + \delta)$$

Using our production function (3) gives

$$\alpha \bar{K}^{\alpha-1} = \rho + \delta$$

Finally, using $\alpha = .33$, $\rho = 1/\beta - 1 = 1/(19/20) - 1 = 20/19 - 19/19 = 1/19$, $\delta = 1/50$, we get

$$\bar{K} = \left(\frac{\frac{33}{100}}{\frac{1}{50} + \frac{1}{19}} \right)^{\frac{67}{100}} \approx 9.57583$$

Let's verify this with Python and then use this steady state $\bar{K}$ as our initial capital stock K_0.

In [6]:
```
ρ = 1 / β - 1
k_ss = f_prime_inv(k=ρ+δ, A=A, α=α)

print(f'steady state for capital is: {k_ss}')
```

```
steady state for capital is: 9.57583816331462
```

Now we plot

In [7]:
```
T = 150
c = np.zeros(T+1)
k = np.zeros(T+2)
c[0] = 0.3
k[0] = k_ss  # Start at steady state
paths = bisection_method(c, k, γ, δ, β, α, A)

plot_paths(paths, ss=k_ss)
```

```
Converged successfully on iteration 39
```

Evidently, in this economy with a large value of T, K_t stays near its initial value at the until the end of time approaches closely.

Evidently, the planner likes the steady state capital stock and wants to stay near there for a long time.

Let's see what happens when we push the initial K_0 below $\bar{K}$.

In [8]:
```
k_init = k_ss / 3   # Below our steady state
T = 150
c = np.zeros(T+1)
k = np.zeros(T+2)
c[0] = 0.3
k[0] = k_init
paths = bisection_method(c, k, γ, δ, β, α, A)

plot_paths(paths, ss=k_ss)
```

```
Converged successfully on iteration 39
```

Notice how the planner pushes capital toward the steady state, stays near there for a while, then pushes K_t toward the terminal value $K_{T+1} = 0$ as t gets close to T.

The following graphs compare outcomes as we vary T.

In [9]:
```
T_list = (150, 75, 50, 25)

fix, axes = plt.subplots(1, 3, figsize=(13, 3))

for T in T_list:
    c = np.zeros(T+1)
    k = np.zeros(T+2)
    c[0] = 0.3
    k[0] = k_init
    paths = bisection_method(c, k, γ, δ, β, α, A)
    plot_paths(paths, ss=k_ss, axes=axes)
```

```
    Converged successfully on iteration 39
Converged successfully on iteration 26
Converged successfully on iteration 25
Converged successfully on iteration 22
```

The following calculation shows that when we set T very large the planner makes the capital stock spend most of its time close to its steady state value.

```
In [10]: T_list = (250, 150, 50, 25)

         fix, axes = plt.subplots(1, 3, figsize=(13, 3))

         for T in T_list:
             c = np.zeros(T+1)
             k = np.zeros(T+2)
             c[0] = 0.3
             k[0] = k_init
             paths = bisection_method(c, k, γ, δ, β, α, A)
             plot_paths(paths, ss=k_ss, axes=axes)
```

```
    Failed to converge and hit maximum iteration
Converged successfully on iteration 39
Converged successfully on iteration 25
Converged successfully on iteration 22
```

The different colors in the above graphs are tied to outcomes with different horizons T.

Notice that as the horizon increases, the planner puts K_t closer to the steady state value $\bar{K}$ for longer.

This pattern reflects a **turnpike** property of the steady state.

A rule of thumb for the planner is

- for whatever K_0 you start with, push K_t toward the steady state and stay there for as long as you can

In loose language: head for the turnpike and stay near it for as long as you can.

As we drive T toward $+\infty$, the planner keeps K_t very close to its steady state for all dates after some transition toward the steady state.

The planner makes the saving rate $\frac{f(K_t)-C_t}{f(K_t)}$ vary over time.

Let's calculate it

In [11]:
```
@njit
def S(K):
    '''Aggregate savings'''
    T = len(K) - 2
    S = np.zeros(T+1)
    for t in range(T+1):
        S[t] = K[t+1] - (1 - δ) * K[t]
    return S

@njit
def s(K):
    '''Savings rate'''
    T = len(K) - 2
    Y = f(A, K, α)
    Y = Y[0:T+1]
    s = S(K) / Y
    return s

def plot_savings(paths, c_ss=None, k_ss=None, s_ss=None, axes=None):

    T = len(paths[0])
    k_star = paths[1]
    savings_path = s(k_star)
    new_paths = (paths[0], paths[1], savings_path)

    if axes is None:
        fix, axes = plt.subplots(1, 3, figsize=(13, 3))

    ylabels = ['$c_t$', '$k_t$', '$s_t$']
    titles = ['Consumption', 'Capital', 'Savings Rate']

    for path, y, title, ax in zip(new_paths, ylabels, titles, axes):
        ax.plot(path)
        ax.set(ylabel=y, title=title, xlabel='t')

    # Plot steady state value of consumption
    if c_ss is not None:
        axes[0].axhline(c_ss, c='k', ls='--', lw=1)

    # Plot steady state value of capital
    if k_ss is not None:
        axes[1].axhline(k_ss, c='k', ls='--', lw=1)

    # Plot steady state value of savings
    if s_ss is not None:
        axes[2].axhline(s_ss, c='k', ls='--', lw=1)
```

```
        axes[1].axvline(T, c='k', ls='--', lw=1)
        axes[1].scatter(T, k_star[-1], s=80)
        plt.tight_layout()

    T_list = (250, 150, 75, 50)

    fix, axes = plt.subplots(1, 3, figsize=(13, 3))

    for T in T_list:
        c = np.zeros(T+1)
        k = np.zeros(T+2)
        c[0] = 0.3
        k[0] = k_init
        paths = bisection_method(c, k, γ, δ, β, α, A)
        plot_savings(paths, k_ss=k_ss, axes=axes)
```

```
    Failed to converge and hit maximum iteration
Converged successfully on iteration 39
Converged successfully on iteration 26
Converged successfully on iteration 25
```

33.3.4 The Limiting Economy

We now consider an economy in which $T = +\infty$.

The appropriate thing to do is to replace terminal condition (10) with

$$\lim_{T \to +\infty} \beta^T u'(C_T) K_{T+1} = 0$$

which is sometimes called a **transversality condition**.

This condition will be satisfied by a path that converges to an optimal steady state.

We can approximate the optimal path from an arbitrary initial K_0 and shooting towards the optimal steady state K at a large but finite $T + 1$.

In the following code, we do this for a large T; we shoot towards the **steady state** and plot consumption, capital and the savings rate.

We know that in the steady state that the saving rate must be fixed and that $\bar{s} = \frac{f(\bar{K}) - \bar{C}}{f(\bar{K})}$.

From (13) the steady state saving rate equals

$$\bar{s} = \frac{\delta \bar{K}}{f(\bar{K})}$$

The steady state savings level $\bar{S} = \bar{s} f(\bar{K})$ is the amount required to offset capital depreciation each period.

We first study optimal capital paths that start below the steady state

In [12]:
```
T = 130

# Steady states
S_ss = δ * k_ss
c_ss = f(A, k_ss, α) - S_ss
s_ss = S_ss / f(A, k_ss, α)

c = np.zeros(T+1)
k = np.zeros(T+2)
c[0] = 0.3
k[0] = k_ss / 3          # Start below steady state
paths = bisection_method(c, k, γ, δ, β, α, A, terminal=k_ss)
plot_savings(paths, k_ss=k_ss, s_ss=s_ss, c_ss=c_ss)
```

```
Converged successfully on iteration 35
```

Since $K_0 < \bar{K}$, $f'(K_0) > \rho + \delta$.

The planner chooses a positive saving rate above the steady state level offsetting depreciation that enables us to increase our capital stock.

Note, $f''(K) < 0$, so as K rises, $f'(K)$ declines.

The planner slowly lowers the savings rate until reaching a steady state where $f'(K) = \rho + \delta$.

33.3.5 Exercise

- Plot the optimal consumption, capital, and savings paths when the initial capital level begins at 1.5 times the steady state level as we shoot towards the steady state at $T = 130$.
- Why does the savings rate respond like it does?

33.3.6 Solution

In [13]:
```
T = 130
```

```
c = np.zeros(T+1)
k = np.zeros(T+2)
c[0] = 0.3
k[0] = k_ss * 1.5   # Start above steady state
paths = bisection_method(c, k, γ, δ, β, α, A, terminal=k_ss)
plot_savings(paths, k_ss=k_ss, s_ss=s_ss, c_ss=c_ss)
```

```
Converged successfully on iteration 31
```

33.4 Competitive Equilibrium

Next, we study a decentralized version of an economy with the same technology and preference structure as our planned economy.

But now there is no planner.

Market prices adjust to reconcile distinct decisions that are made separately by a representative household and a representative firm.

The technology for producing goods and accumulating capital via physical investment remains as in our planned economy.

There is a representative consumer who has the same preferences over consumption plans as did the consumer in the planned economy.

Instead of being told what to consume and save by a planner, the household chooses for itself subject to a budget constraint

- At each time t, the household receives wages and rentals of capital from a firm – these comprise its **income** at time t.
- The consumer decides how much income to allocate to consumption or to savings.
- The household can save either by acquiring additional physical capital (it trades one for one with time t consumption) or by acquiring claims on consumption at dates other than t.
- A utility-maximizing household owns all physical capital and labor and rents them to the firm.
- The household consumes, supplies labor, and invests in physical capital.
- A profit-maximizing representative firm operates the production technology.
- The firm rents labor and capital each period from the representative household and sells its output each period to the household.
- The representative household and the representative firm are both **price takers:**
  - they (correctly) believe that prices are not affected by their choices

Note: We are free to think of there being a large number M of identical representative consumers and M identical representative firms.

33.4.1 Firm Problem

At time t the representative firm hires labor $\tilde{n}_t$ and capital $\tilde{k}_t$.

The firm's profits at time t are

$$F(\tilde{k}_t, \tilde{n}_t) - w_t \tilde{n}_t - \eta_t \tilde{k}_t$$

where w_t is a wage rate at t and η_t is the rental rate on capital at t.

As in the planned economy model

$$F(\tilde{k}_t, \tilde{n}_t) = A \tilde{k}_t^{\alpha} \tilde{n}_t^{1-\alpha}$$

Zero Profit Conditions

Zero-profits condition for capital and labor are

$$F_k(\tilde{k}_t, \tilde{n}_t) = \eta_t$$

and

$$F_n(\tilde{k}_t, \tilde{n}_t) = w_t \tag{14}$$

These conditions emerge from a no-arbitrage requirement.

To describe this line of reasoning, we begin by applying a theorem of Euler about linearly homogenous functions.

The theorem applies to the Cobb-Douglas production function because it assumed displays constant returns to scale:

$$\alpha F(\tilde{k}_t, \tilde{n}_t) = F(\alpha \tilde{k}_t, \alpha \tilde{n}_t)$$

for $\alpha \in (0, 1)$.

Taking the partial derivative $\frac{\partial F}{\partial \alpha}$ on both sides of the above equation gives

$$F(\tilde{k}_t, \tilde{n}_t) =_{\text{chain rule}} \frac{\partial F}{\partial \tilde{k}_t} \tilde{k}_t + \frac{\partial F}{\partial \tilde{n}_t} \tilde{n}_t$$

Rewrite the firm's profits as

$$\frac{\partial F}{\partial \tilde{k}_t} \tilde{k}_t + \frac{\partial F}{\partial \tilde{n}_t} \tilde{n}_t - w_t \tilde{n}_t - \eta_t k_t$$

or

$$\left(\frac{\partial F}{\partial \tilde{k}_t} - \eta_t\right)\tilde{k}_t + \left(\frac{\partial F}{\partial \tilde{n}_t} - w_t\right)\tilde{n}_t$$

Because F is homogeneous of degree 1, it follows that $\frac{\partial F}{\partial \tilde{k}_t}$ and $\frac{\partial F}{\partial \tilde{n}_t}$ are homogeneous of degree 0 and therefore fixed with respect to $\tilde{k}_t$ and $\tilde{n}_t$.

If $\frac{\partial F}{\partial \tilde{k}_t} > \eta_t$, then the firm makes positive profits on each additional unit of $\tilde{k}_t$, so it will want to make $\tilde{k}_t$ arbitrarily large.

But setting $\tilde{k}_t = +\infty$ is not physically feasible, so presumably **equilibrium** prices will assume values that present the firm with no such arbitrage opportunity.

A related argument applies if $\frac{\partial F}{\partial \tilde{n}_t} > w_t$.

If $\frac{\partial \tilde{k}_t}{\partial \tilde{k}_t} < \eta_t$, the firm will set $\tilde{k}_t$ to zero.

Again, **equilibrium** prices won't incentive the firm to do that.

And so on…

It is convenient to define $\vec{w}_t = \{w_0, \dots, w_T\}$and $\vec{\eta}_t = \{\eta_0, \dots, \eta_T\}$.

33.4.2 Household Problem

A representative household lives at $t = 0, 1, \dots, T$.

At t, the household rents 1 unit of labor and k_t units of capital to a firm and receives income

$$w_t 1 + \eta_t k_t$$

At t the household allocates its income to the following purchases

$$(c_t + (k_{t+1} - (1-\delta)k_t))$$

Here $(k_{t+1} - (1-\delta)k_t)$ is the household's net investment in physical capital and $\delta \in (0,1)$ is again a depreciation rate of capital.

In period t is free to purchase more goods to be consumed and invested in physical capital than its income from supplying capital and labor to the firm, provided that in some other periods its income exceeds its purchases.

A household's net excess demand for time t consumption goods is the gap

$$e_t \equiv (c_t + (k_{t+1} - (1-\delta)k_t)) - (w_t 1 + \eta_t k_t)$$

Let $\vec{c} = \{c_0, \dots, c_T\}$ and let $\vec{k} = \{k_1, \dots, k_{T+1}\}$.

k_0 is given to the household.

33.4.3 Market Structure for Intertemporal Trades

There is a **single** grand competitive market in which a representative household can trade date 0 goods for goods at all other dates $t = 1, 2, \dots, T$.

What matters are not **bilateral** trades of the good at one date t for the good at another date $\tilde{t} \neq t$.

Instead, think of there being **multilateral** and **multitemporal** trades in which bundles of goods at some dates can be traded for bundles of goods at some other dates.

There exist **complete markets** in such bundles with associated market prices.

33.4.4 Market Prices

Let q_t^0 be the price of a good at date t relative to a good at date 0.

$\{q_t^0\}_{t=0}^T$ is a vector of **Hicks-Arrow prices**, named after the 1972 joint economics Nobel prize winners who used such prices in some of their important work.

Evidently,

$$q_t^0 = \frac{\text{number of time 0 goods}}{\text{number of time t goods}}$$

Because q_t^0 is a **relative price**, the units in terms of which prices are quoted are arbitrary – we can normalize them without substantial consequence.

If we use the price vector $\{q_t^0\}_{t=0}^T$ to evaluate a stream of excess demands $\{e_t\}_{t=0}^T$ we compute the **present value** of $\{e_t\}_{t=0}^T$ to be $\sum_{t=0}^T q_t^0 e_t$.

That the market is **multitemporal** is reflected in the situation that the household faces a **single** budget constraint.

It states that the present value of the household's net excess demands must be zero:

$$\sum_{t=0}^T q_t^0 e_t \leq 0$$

or

$$\sum_{t=0}^T q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t) - (w_t 1 + \eta_t k_t)\right) \leq 0$$

33.4.5 Household Problem

The household faces the constrained optimization problem:

$$\begin{aligned} &\max_{\vec{c},\vec{k}} \sum_{t=0}^T \beta^t u(c_t) \\ \text{subject to } &\sum_{t=0}^T q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t) - w_t - \eta_t k_t\right) \leq 0 \end{aligned}$$

33.4.6 Definitions

- A **price system** is a sequence $\{q_t^0, \eta_t, w_t\}_{t=0}^T = \{\vec{q}, \vec{\eta}, \vec{w}\}$.
- An **allocation** is a sequence $\{c_t, k_{t+1}, n_t = 1\}_{t=0}^T = \{\vec{c}, \vec{k}, \vec{n} = 1\}$.
- A **competitive equilibrium** is a price system and an allocation for which
  - Given the price system, the allocation solves the household's problem.
  - Given the price system, the allocation solves the firm's problem.

33.4.7 Computing a Competitive Equilibrium

We shall compute a competitive equilibrium using a **guess and verify** approach.

- We shall **guess** equilibrium price sequences $\{\vec{q}, \vec{\eta}, \vec{w}\}$.
- We shall then **verify** that at those prices, the household and the firm choose the same allocation.

Guess for Price System

We have computed an allocation $\{\vec{C}, \vec{K}, \vec{1}\}$ that solves the planning problem.

We use that allocation to construct our guess for the equilibrium price system.

In particular, we guess that for $t = 0, \dots, T$:

$$\lambda q_t^0 = \beta^t u'(K_t) = \beta^t \mu_t \tag{15}$$

$$w_t = f(K_t) - K_t f'(K_t) \tag{16}$$

$$\eta_t = f'(K_t) \tag{17}$$

At these prices, let the capital chosen by the household be

$$k_t^*(\vec{q}, \vec{w}, \vec{\eta}), \quad t \geq 0 \tag{18}$$

and let the allocation chosen by the firm be

$$\tilde{k}_t^*(\vec{q}, \vec{w}, \vec{\eta}), \quad t \geq 0$$

and so on.

If our guess for the equilibrium price system is correct, then it must occur that

$$k_t^* = \tilde{k}_t^* \tag{19}$$

$$1 = \tilde{n}_t^* \tag{20}$$

$$c_t^* + k_{t+1}^* - (1-\delta)k_t^* = F(\tilde{k}_t^*, \tilde{n}_t^*)$$

We shall verify that for $t = 0, \dots, T$ the allocations chosen by the household and the firm both equal the allocation that solves the planning problem:

$$k_t^* = \tilde{k}_t^* = K_t, \tilde{n}_t = 1, c_t^* = C_t \tag{21}$$

33.4.8 Verification Procedure

Our approach is to stare at first-order necessary conditions for the optimization problems of the household and the firm.

At the price system we have guessed, both sets of first-order conditions are satisfied at the allocation that solves the planning problem.

33.4.9 Household's Lagrangian

To solve the household's problem, we formulate the appropriate Lagrangian and pose the min-max problem:

$$\min_{\lambda} \max_{\vec{c},\vec{k}} \mathcal{L}(\vec{c}, \vec{k}, \lambda) = \sum_{t=0}^{T} \beta^t u(c_t) + \lambda \left(\sum_{t=0}^{T} q_t^0 \left(((1-\delta)k_t - w_t) + \eta_t k_t - c_t - k_{t+1} \right) \right)$$

First-order conditions are

$$c_t : \quad \beta^t u'(c_t) - \lambda q_t^0 = 0 \quad t = 0, 1, \dots, T \tag{22}$$

$$k_t : \quad -\lambda q_t^0 \left[(1-\delta) + \eta_t \right] + \lambda q_{t-1}^0 = 0 \quad t = 1, 2, \dots, T+1 \tag{23}$$

$$\lambda : \quad \left(\sum_{t=0}^{T} q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t) - w_t - \eta_t k_t \right) \right) \leq 0 \tag{24}$$

$$k_{T+1} : \quad -\lambda q_0^{T+1} \leq 0, \ \leq 0 \text{ if } k_{T+1} = 0; \ = 0 \text{ if } k_{T+1} > 0 \tag{25}$$

Now we plug in for our guesses of prices and derive all the FONC of the planner problem (7)-(10):

Combining (22) and (15), we get:

$$u'(C_t) = \mu_t$$

which is (7).

Combining (23), (15), and (17) we get:

$$-\lambda \beta^t \mu_t \left[(1-\delta) + f'(K_t) \right] + \lambda \beta^{t-1} \mu_{t-1} = 0 \tag{26}$$

Rewriting (26) by dividing by λ on both sides (which is nonzero due to u'>0) we get:

$$\beta^t \mu_t[(1-\delta+f'(K_t)] = \beta^{t-1}\mu_{t-1}$$

or

$$\beta \mu_t[(1-\delta+f'(K_t)] = \mu_{t-1}$$

which is (8).

Combining (24), (15), (16) and (17) after multiplying both sides of (24) by λ, we get:

$$\sum_{t=0}^{T} \beta^t \mu_t \left(C_t + (K_{t+1} - (1-\delta)K_t) - f(K_t) + K_t f'(K_t) - f'(K_t)K_t\right) \leq 0$$

Cancelling,

$$\sum_{t=0}^{T} \beta^t \mu_t \left(C_t + K_{t+1} - (1-\delta)K_t - F(K_t, 1)\right) \leq 0$$

Since β^t and μ_t are always positive here, (excepting perhaps the T+1 period) we get:

$$C_t + K_{t+1} - (1-\delta)K_t - F(K_t, 1) = 0 \quad \text{for all } t \text{ in } 0, \dots, T$$

which is (9).

Combining (25) and (15), we get:

$$-\beta^{T+1}\mu_{T+1} \leq 0$$

Dividing both sides by β^{T+1} which will be strictly positive here, we get:

$$-\mu_{T+1} \leq 0$$

which is the (10) of our planning problem.

Thus, at our guess of the equilibrium price system, the allocation that solves the planning problem also solves the problem faced by a representative household living in a competitive equilibrium.

We now consider the problem faced by a firm in a competitive equilibrium:

If we plug in (21) into (14) for all t, we get

$$\frac{\partial F(K_t, 1)}{\partial K_t} = f'(K_t) = \eta_t$$

which is (17).

If we now plug (21) into (14) for all t, we get:

$$\frac{\partial F(\tilde{K}_t, 1)}{\partial \tilde{L}} = f(K_t) - f'(K_t)K_t = w_t$$

which is exactly (18).

Thus, at our guess of the equilibrium price system, the allocation that solves the planning problem also solves the problem faced by a firm within a competitive equilibrium.

By (19) and (20) this allocation is identical to the one that solves the consumer's problem.

Note: Because budget sets are affected only by relative prices, $\{q_0^t\}$ is determined only up to multiplication by a positive constant.

Normalization: We are free to choose a $\{q_0^t\}$ that makes $\lambda = 1$, thereby making q_0^t be measured in units of the marginal utility of time 0 goods.

We will also plot q, w and η below to show the prices that induce the same aggregate movements we saw earlier in the planning problem.

In [14]:
```
@njit
def q_func(β, c, γ):
    # Here we choose numeraire to be u'(c_0) -- this is q^(t_0)_t
    T = len(c) - 2
    q = np.zeros(T+1)
    q[0] = 1
    for t in range(1, T+2):
        q[t] = β**t * u_prime(c[t], γ)
    return q

@njit
def w_func(A, k, α):
    w = f(A, k, α) - k * f_prime(A, k, α)
    return w

@njit
def η_func(A, k, α):
    η = f_prime(A, k, α)
    return η
```

Now we calculate and plot for each T

In [15]:
```
T_list = (250, 150, 75, 50)

fix, axes = plt.subplots(2, 3, figsize=(13, 6))
titles = ['Arrow-Hicks Prices', 'Labor Rental Rate', 'Capital Rental Rate',
          'Consumption', 'Capital', 'Lagrange Multiplier']
ylabels = ['$q_t^0$', '$w_t$', '$\eta_t$', '$c_t$', '$k_t$', '$\mu_t$']

for T in T_list:
    c = np.zeros(T+1)
    k = np.zeros(T+2)
    c[0] = 0.3
    k[0] = k_ss / 3
    c, k, μ = bisection_method(c, k, γ, δ, β, α, A)

    q = q_func(β, c, γ)
    w = w_func(β, k, α)[:-1]
    η = η_func(A, k, α)[:-1]
    plots = [q, w, η, c, k, μ]
```

```
    for ax, plot, title, y in zip(axes.flatten(), plots, titles, ylabels):
        ax.plot(plot)
        ax.set(title=title, ylabel=y, xlabel='t')
        if title is 'Capital':
            ax.axhline(k_ss, lw=1, ls='--', c='k')
        if title is 'Consumption':
            ax.axhline(c_ss, lw=1, ls='--', c='k')


plt.tight_layout()
plt.show()
```

```
  Failed to converge and hit maximum iteration
Converged successfully on iteration 39
Converged successfully on iteration 26
Converged successfully on iteration 25
```

Varying Curvature

Now we see how our results change if we keep T constant, but allow the curvature parameter, γ to vary, starting with K_0 below the steady state.

We plot the results for $T = 150$

```
In [16]: γ_list = (1.1, 4, 6, 8)
         T = 150

         fix, axes = plt.subplots(2, 3, figsize=(13, 6))

         for γ in γ_list:
             c = np.zeros(T+1)
             k = np.zeros(T+2)
             c[0] = 0.3
             k[0] = k_ss / 3
             c, k, μ = bisection_method(c, k, γ, δ, β, α, A)
```

```
    q = q_func(β, c, γ)
    w = w_func(β, k, α)[:-1]
    η = η_func(A, k, α)[:-1]
    plots = [q, w, η, c, k, μ]

    for ax, plot, title, y in zip(axes.flatten(), plots, titles, ylabels):
        ax.plot(plot, label=f'$\gamma = {γ}$')
        ax.set(title=title, ylabel=y, xlabel='t')
        if title is 'Capital':
            ax.axhline(k_ss, lw=1, ls='--', c='k')
        if title is 'Consumption':
            ax.axhline(c_ss, lw=1, ls='--', c='k')

axes[0, 0].legend()
plt.tight_layout()
plt.show()
```

```
   Converged successfully on iteration 44
Converged successfully on iteration 37
Converged successfully on iteration 37
Converged successfully on iteration 37
```

Adjusting γ means adjusting how much individuals prefer to smooth consumption.

Higher γ means individuals prefer to smooth more resulting in slower adjustments to the steady state allocations.

Vice-versa for lower γ.

33.4.10 Yield Curves and Hicks-Arrow Prices Again

Now, we compute Hicks-Arrow prices again, but also calculate the implied yields to maturity.

This will let us plot a **yield curve**.

The key formulas are:

The **yield to maturity**

$$r_{t_0,t} = -\frac{\log q_t^{t_0}}{t - t_0}$$

A generic Hicks-Arrow price for any base-year $t_0 \leq t$

$$q_t^{t_0} = \beta^{t-t_0} \frac{u'(c_t)}{u'(c_{t_0})} = \beta^{t-t_0} \frac{c_t^{-\gamma}}{c_{t_0}^{-\gamma}}$$

We redefine our function for q to allow arbitrary base years, and define a new function for r, then plot both.

First, we plot when $t_0 = 0$ as before, for different values of T, with K_0 below the steady state

In [17]:
```
@njit
def q_func(t_0, β, c, γ):
    # Here we choose numeraire to be u'(c_0) -- this is q^(t_0)_t
    T = len(c)
    q = np.zeros(T+1-t_0)
    q[0] = 1
    for t in range(t_0+1, T):
        q[t-t_0] = β**(t - t_0) * u_prime(c[t], γ) / u_prime(c[t_0], γ)
    return q

@njit
def r_func(t_0, β, c, γ):
    '''Yield to maturity'''
    T = len(c) - 1
    r = np.zeros(T+1-t_0)
    for t in range(t_0+1, T+1):
        r[t-t_0]= -np.log(q_func(t_0, β, c, γ)[t-t_0]) / (t - t_0)
    return r

t_0 = 0
T_list = [150, 75, 50]
γ = 2
titles = ['Hicks-Arrow Prices', 'Yields']
ylabels = ['$q_t^0$', '$r_t^0$']

fig, axes = plt.subplots(1, 2, figsize=(10, 5))

for T in T_list:
    c = np.zeros(T+1)
    k = np.zeros(T+2)
    c[0] = 0.3
    k[0] = k_ss / 3
    c, k, μ = bisection_method(c, k, γ, δ, β, α, A)
    q = q_func(t_0, β, c, γ)
    r = r_func(t_0, β, c, γ)

    for ax, plot, title, y in zip(axes, (q, r), titles, ylabels):
        ax.plot(plot)
        ax.set(title=title, ylabel=y, xlabel='t')

plt.tight_layout()
plt.show()
```

```
    Converged successfully on iteration 39
Converged successfully on iteration 26
Converged successfully on iteration 25
```

Now we plot when $t_0 = 20$

```
In [18]: t_0 = 20

         fig, axes = plt.subplots(1, 2, figsize=(10, 5))

         for T in T_list:
             c = np.zeros(T+1)
             k = np.zeros(T+2)
             c[0] = 0.3
             k[0] = k_ss / 3
             c, k, μ = bisection_method(c, k, γ, δ, β, α, A)
             q = q_func(t_0, β, c, γ)
             r = r_func(t_0, β, c, γ)

             for ax, plot, title, y in zip(axes, (q, r), titles, ylabels):
                 ax.plot(plot)
                 ax.set(title=title, ylabel=y, xlabel='t')

         axes[1].set_title(f'Yields at $t_0 = {t_0}$')
         plt.tight_layout()
         plt.show()
```

```
    Converged successfully on iteration 39
Converged successfully on iteration 26
Converged successfully on iteration 25
```

We shall have more to say about the term structure of interest rates in a later lecture on the topic.

Chapter 34

A First Look at the Kalman Filter

34.1 Contents

- Overview 34.2
- The Basic Idea 34.3
- Convergence 34.4
- Implementation 34.5
- Exercises 34.6
- Solutions 34.7

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

```
In [1]: !pip install --upgrade quantecon
```

34.2 Overview

This lecture provides a simple and intuitive introduction to the Kalman filter, for those who either

- have heard of the Kalman filter but don't know how it works, or
- know the Kalman filter equations, but don't know where they come from

For additional (more advanced) reading on the Kalman filter, see

- [108], section 2.7
- [8]

The second reference presents a comprehensive treatment of the Kalman filter.

Required knowledge: Familiarity with matrix manipulations, multivariate normal distributions, covariance matrices, etc.

We'll need the following imports:

```
In [2]: from scipy import linalg
        import numpy as np
        import matplotlib.cm as cm
        import matplotlib.pyplot as plt
        %matplotlib inline
        from quantecon import Kalman, LinearStateSpace
```

```
from scipy.stats import norm
from scipy.integrate import quad
from numpy.random import multivariate_normal
from scipy.linalg import eigvals
```

34.3 The Basic Idea

The Kalman filter has many applications in economics, but for now let's pretend that we are rocket scientists.

A missile has been launched from country Y and our mission is to track it.

Let $x \in \mathbb{R}^2$ denote the current location of the missile—a pair indicating latitude-longitude coordinates on a map.

At the present moment in time, the precise location x is unknown, but we do have some beliefs about x.

One way to summarize our knowledge is a point prediction $\hat{x}$

- But what if the President wants to know the probability that the missile is currently over the Sea of Japan?
- Then it is better to summarize our initial beliefs with a bivariate probability density p
  - $\int_E p(x)dx$ indicates the probability that we attach to the missile being in region E.

The density p is called our *prior* for the random variable x.

To keep things tractable in our example, we assume that our prior is Gaussian.

In particular, we take

$$p = N(\hat{x}, \Sigma) \tag{1}$$

where $\hat{x}$ is the mean of the distribution and Σ is a 2×2 covariance matrix. In our simulations, we will suppose that

$$\hat{x} = \begin{pmatrix} 0.2 \\ -0.2 \end{pmatrix}, \qquad \Sigma = \begin{pmatrix} 0.4 & 0.3 \\ 0.3 & 0.45 \end{pmatrix} \tag{2}$$

This density $p(x)$ is shown below as a contour map, with the center of the red ellipse being equal to $\hat{x}$.

In [3]:
```
# Set up the Gaussian prior density p
Σ = [[0.4, 0.3], [0.3, 0.45]]
Σ = np.matrix(Σ)
x_hat = np.matrix([0.2, -0.2]).T
# Define the matrices G and R from the equation y = G x + N(0, R)
G = [[1, 0], [0, 1]]
G = np.matrix(G)
R = 0.5 * Σ
# The matrices A and Q
A = [[1.2, 0], [0, -0.2]]
A = np.matrix(A)
Q = 0.3 * Σ
# The observed value of y
```

```
y = np.matrix([2.3, -1.9]).T

# Set up grid for plotting
x_grid = np.linspace(-1.5, 2.9, 100)
y_grid = np.linspace(-3.1, 1.7, 100)
X, Y = np.meshgrid(x_grid, y_grid)

def bivariate_normal(x, y, σ_x=1.0, σ_y=1.0, μ_x=0.0, μ_y=0.0, σ_xy=0.0):
    """
    Compute and return the probability density function of bivariate normal
    distribution of normal random variables x and y

    Parameters
    ----------
    x : array_like(float)
        Random variable

    y : array_like(float)
        Random variable

    σ_x : array_like(float)
          Standard deviation of random variable x

    σ_y : array_like(float)
          Standard deviation of random variable y

    μ_x : scalar(float)
          Mean value of random variable x

    μ_y : scalar(float)
          Mean value of random variable y

    σ_xy : array_like(float)
           Covariance of random variables x and y

    """

    x_μ = x - μ_x
    y_μ = y - μ_y

    ρ = σ_xy / (σ_x * σ_y)
    z = x_μ**2 / σ_x**2 + y_μ**2 / σ_y**2 - 2 * ρ * x_μ * y_μ / (σ_x * σ_y)
    denom = 2 * np.pi * σ_x * σ_y * np.sqrt(1 - ρ**2)
    return np.exp(-z / (2 * (1 - ρ**2))) / denom

def gen_gaussian_plot_vals(μ, C):
    "Z values for plotting the bivariate Gaussian N(μ, C)"
    m_x, m_y = float(μ[0]), float(μ[1])
    s_x, s_y = np.sqrt(C[0, 0]), np.sqrt(C[1, 1])
    s_xy = C[0, 1]
    return bivariate_normal(X, Y, s_x, s_y, m_x, m_y, s_xy)

# Plot the figure

fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()

Z = gen_gaussian_plot_vals(x_hat, Σ)
```

```
ax.contourf(X, Y, Z, 6, alpha=0.6, cmap=cm.jet)
cs = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs, inline=1, fontsize=10)

plt.show()
```

34.3.1 The Filtering Step

We are now presented with some good news and some bad news.

The good news is that the missile has been located by our sensors, which report that the current location is $y = (2.3, -1.9)$.

The next figure shows the original prior $p(x)$ and the new reported location y

```
In [4]: fig, ax = plt.subplots(figsize=(10, 8))
        ax.grid()

        Z = gen_gaussian_plot_vals(x_hat, Σ)
        ax.contourf(X, Y, Z, 6, alpha=0.6, cmap=cm.jet)
        cs = ax.contour(X, Y, Z, 6, colors="black")
        ax.clabel(cs, inline=1, fontsize=10)
        ax.text(float(y[0]), float(y[1]), "$y$", fontsize=20, color="black")

        plt.show()
```

The bad news is that our sensors are imprecise.

In particular, we should interpret the output of our sensor not as $y = x$, but rather as

$$y = Gx + v, \quad \text{where} \quad v \sim N(0, R) \tag{3}$$

Here G and R are 2×2 matrices with R positive definite. Both are assumed known, and the noise term v is assumed to be independent of x.

How then should we combine our prior $p(x) = N(\hat{x}, \Sigma)$ and this new information y to improve our understanding of the location of the missile?

As you may have guessed, the answer is to use Bayes' theorem, which tells us to update our prior $p(x)$ to $p(x \,|\, y)$ via

$$p(x \,|\, y) = \frac{p(y \,|\, x)\, p(x)}{p(y)}$$

where $p(y) = \int p(y \,|\, x)\, p(x) dx$.

In solving for $p(x \,|\, y)$, we observe that

- $p(x) = N(\hat{x}, \Sigma)$.
- In view of (3), the conditional density $p(y \,|\, x)$ is $N(Gx, R)$.
- $p(y)$ does not depend on x, and enters into the calculations only as a normalizing constant.

Because we are in a linear and Gaussian framework, the updated density can be computed by calculating population linear regressions.

In particular, the solution is known Section **??** to be

$$p(x \,|\, y) = N(\hat{x}^F, \Sigma^F)$$

where

$$\hat{x}^F := \hat{x} + \Sigma G'(G\Sigma G' + R)^{-1}(y - G\hat{x}) \quad \text{and} \quad \Sigma^F := \Sigma - \Sigma G'(G\Sigma G' + R)^{-1} G\Sigma \tag{4}$$

Here $\Sigma G'(G\Sigma G' + R)^{-1}$ is the matrix of population regression coefficients of the hidden object $x - \hat{x}$ on the surprise $y - G\hat{x}$.

This new density $p(x \,|\, y) = N(\hat{x}^F, \Sigma^F)$ is shown in the next figure via contour lines and the color map.

The original density is left in as contour lines for comparison

In [5]:
```
fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()

Z = gen_gaussian_plot_vals(x_hat, Σ)
cs1 = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs1, inline=1, fontsize=10)
M = Σ * G.T * linalg.inv(G * Σ * G.T + R)
x_hat_F = x_hat + M * (y - G * x_hat)
Σ_F = Σ - M * G * Σ
new_Z = gen_gaussian_plot_vals(x_hat_F, Σ_F)
cs2 = ax.contour(X, Y, new_Z, 6, colors="black")
ax.clabel(cs2, inline=1, fontsize=10)
ax.contourf(X, Y, new_Z, 6, alpha=0.6, cmap=cm.jet)
ax.text(float(y[0]), float(y[1]), "$y$", fontsize=20, color="black")

plt.show()
```

Our new density twists the prior $p(x)$ in a direction determined by the new information $y - G\hat{x}$.

In generating the figure, we set G to the identity matrix and $R = 0.5\Sigma$ for Σ defined in (2).

34.3.2 The Forecast Step

What have we achieved so far?

We have obtained probabilities for the current location of the state (missile) given prior and current information.

This is called "filtering" rather than forecasting because we are filtering out noise rather than looking into the future.

- $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ is called the *filtering distribution*

But now let's suppose that we are given another task: to predict the location of the missile after one unit of time (whatever that may be) has elapsed.

To do this we need a model of how the state evolves.

Let's suppose that we have one, and that it's linear and Gaussian. In particular,

$$x_{t+1} = Ax_t + w_{t+1}, \quad \text{where} \quad w_t \sim N(0, Q) \tag{5}$$

Our aim is to combine this law of motion and our current distribution $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ to come up with a new *predictive* distribution for the location in one unit of time.

In view of (5), all we have to do is introduce a random vector $x^F \sim N(\hat{x}^F, \Sigma^F)$ and work out the distribution of $Ax^F + w$ where w is independent of x^F and has distribution $N(0, Q)$.

Since linear combinations of Gaussians are Gaussian, $Ax^F + w$ is Gaussian.

Elementary calculations and the expressions in (4) tell us that

$$\mathbb{E}[Ax^F + w] = A\mathbb{E}x^F + \mathbb{E}w = A\hat{x}^F = A\hat{x} + A\Sigma G'(G\Sigma G' + R)^{-1}(y - G\hat{x})$$

and

$$\operatorname{Var}[Ax^F + w] = A \operatorname{Var}[x^F]A' + Q = A\Sigma^F A' + Q = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q$$

The matrix $A\Sigma G'(G\Sigma G' + R)^{-1}$ is often written as K_Σ and called the *Kalman gain*.

- The subscript Σ has been added to remind us that K_Σ depends on Σ, but not y or $\hat{x}$.

Using this notation, we can summarize our results as follows.

Our updated prediction is the density $N(\hat{x}_{new}, \Sigma_{new})$ where

$$\begin{aligned} \hat{x}_{new} &:= A\hat{x} + K_\Sigma(y - G\hat{x}) \\ \Sigma_{new} &:= A\Sigma A' - K_\Sigma G\Sigma A' + Q \end{aligned} \tag{6}$$

- The density $p_{new}(x) = N(\hat{x}_{new}, \Sigma_{new})$ is called the *predictive distribution*

The predictive distribution is the new density shown in the following figure, where the update has used parameters.

$$A = \begin{pmatrix} 1.2 & 0.0 \\ 0.0 & -0.2 \end{pmatrix}, \qquad Q = 0.3 * \Sigma$$

In [6]:
```
fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()

# Density 1
Z = gen_gaussian_plot_vals(x_hat, Σ)
cs1 = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs1, inline=1, fontsize=10)

# Density 2
M = Σ * G.T * linalg.inv(G * Σ * G.T + R)
x_hat_F = x_hat + M * (y - G * x_hat)
Σ_F = Σ - M * G * Σ
Z_F = gen_gaussian_plot_vals(x_hat_F, Σ_F)
cs2 = ax.contour(X, Y, Z_F, 6, colors="black")
ax.clabel(cs2, inline=1, fontsize=10)

# Density 3
new_x_hat = A * x_hat_F
new_Σ = A * Σ_F * A.T + Q
new_Z = gen_gaussian_plot_vals(new_x_hat, new_Σ)
cs3 = ax.contour(X, Y, new_Z, 6, colors="black")
ax.clabel(cs3, inline=1, fontsize=10)
ax.contourf(X, Y, new_Z, 6, alpha=0.6, cmap=cm.jet)
```

```
ax.text(float(y[0]), float(y[1]), "$y$", fontsize=20, color="black")

plt.show()
```

34.3.3 The Recursive Procedure

Let's look back at what we've done.

We started the current period with a prior $p(x)$ for the location x of the missile.

We then used the current measurement y to update to $p(x \,|\, y)$.

Finally, we used the law of motion (5) for $\{x_t\}$ to update to $p_{new}(x)$.

If we now step into the next period, we are ready to go round again, taking $p_{new}(x)$ as the current prior.

Swapping notation $p_t(x)$ for $p(x)$ and $p_{t+1}(x)$ for $p_{new}(x)$, the full recursive procedure is:

1. Start the current period with prior $p_t(x) = N(\hat{x}_t, \Sigma_t)$.
2. Observe current measurement y_t.
3. Compute the filtering distribution $p_t(x \,|\, y) = N(\hat{x}_t^F, \Sigma_t^F)$ from $p_t(x)$ and y_t, applying Bayes rule and the conditional distribution (3).
4. Compute the predictive distribution $p_{t+1}(x) = N(\hat{x}_{t+1}, \Sigma_{t+1})$ from the filtering distribution and (5).

5. Increment t by one and go to step 1.

Repeating (6), the dynamics for $\hat{x}_t$ and Σ_t are as follows

$$\begin{aligned} \hat{x}_{t+1} &= A\hat{x}_t + K_{\Sigma_t}(y_t - G\hat{x}_t) \\ \Sigma_{t+1} &= A\Sigma_t A' - K_{\Sigma_t} G\Sigma_t A' + Q \end{aligned} \tag{7}$$

These are the standard dynamic equations for the Kalman filter (see, for example, [108], page 58).

34.4 Convergence

The matrix Σ_t is a measure of the uncertainty of our prediction $\hat{x}_t$ of x_t.

Apart from special cases, this uncertainty will never be fully resolved, regardless of how much time elapses.

One reason is that our prediction $\hat{x}_t$ is made based on information available at $t-1$, not t.

Even if we know the precise value of x_{t-1} (which we don't), the transition equation (5) implies that $x_t = Ax_{t-1} + w_t$.

Since the shock w_t is not observable at $t-1$, any time $t-1$ prediction of x_t will incur some error (unless w_t is degenerate).

However, it is certainly possible that Σ_t converges to a constant matrix as $t \to \infty$.

To study this topic, let's expand the second equation in (7):

$$\Sigma_{t+1} = A\Sigma_t A' - A\Sigma_t G'(G\Sigma_t G' + R)^{-1} G\Sigma_t A' + Q \tag{8}$$

This is a nonlinear difference equation in Σ_t.

A fixed point of (8) is a constant matrix Σ such that

$$\Sigma = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1} G\Sigma A' + Q \tag{9}$$

Equation (8) is known as a discrete-time Riccati difference equation.

Equation (9) is known as a discrete-time algebraic Riccati equation.

Conditions under which a fixed point exists and the sequence $\{\Sigma_t\}$ converges to it are discussed in [9] and [8], chapter 4.

A sufficient (but not necessary) condition is that all the eigenvalues λ_i of A satisfy $|\lambda_i| < 1$ (cf. e.g., [8], p. 77).

(This strong condition assures that the unconditional distribution of x_t converges as $t \to +\infty$)

In this case, for any initial choice of Σ_0 that is both non-negative and symmetric, the sequence $\{\Sigma_t\}$ in (8) converges to a non-negative symmetric matrix Σ that solves (9).

34.5 Implementation

The class `Kalman` from the QuantEcon.py package implements the Kalman filter

- Instance data consists of:
  - the moments $(\hat{x}_t, \Sigma_t)$ of the current prior.
  - An instance of the LinearStateSpace class from QuantEcon.py.

The latter represents a linear state space model of the form

$$
\begin{aligned}
x_{t+1} &= Ax_t + Cw_{t+1} \\
y_t &= Gx_t + Hv_t
\end{aligned}
$$

where the shocks w_t and v_t are IID standard normals.

To connect this with the notation of this lecture we set

$$
Q := CC' \quad \text{and} \quad R := HH'
$$

- The class `Kalman` from the QuantEcon.py package has a number of methods, some that we will wait to use until we study more advanced applications in subsequent lectures.
- Methods pertinent for this lecture are:
  - `prior_to_filtered`, which updates $(\hat{x}_t, \Sigma_t)$ to $(\hat{x}_t^F, \Sigma_t^F)$
  - `filtered_to_forecast`, which updates the filtering distribution to the predictive distribution – which becomes the new prior $(\hat{x}_{t+1}, \Sigma_{t+1})$
  - `update`, which combines the last two methods
  - a `stationary_values`, which computes the solution to (9) and the corresponding (stationary) Kalman gain

You can view the program on GitHub.

34.6 Exercises

34.6.1 Exercise 1

Consider the following simple application of the Kalman filter, loosely based on [108], section 2.9.2.

Suppose that

- all variables are scalars
- the hidden state $\{x_t\}$ is in fact constant, equal to some $\theta \in \mathbb{R}$ unknown to the modeler

State dynamics are therefore given by (5) with $A = 1$, $Q = 0$ and $x_0 = \theta$.

The measurement equation is $y_t = \theta + v_t$ where v_t is $N(0, 1)$ and IID.

The task of this exercise to simulate the model and, using the code from `kalman.py`, plot the first five predictive densities $p_t(x) = N(\hat{x}_t, \Sigma_t)$.

As shown in [108], sections 2.9.1–2.9.2, these distributions asymptotically put all mass on the unknown value θ.

In the simulation, take $\theta = 10$, $\hat{x}_0 = 8$ and $\Sigma_0 = 1$.

Your figure should – modulo randomness – look something like this

34.6.2 Exercise 2

The preceding figure gives some support to the idea that probability mass converges to θ.

To get a better idea, choose a small $\epsilon > 0$ and calculate

$$z_t := 1 - \int_{\theta-\epsilon}^{\theta+\epsilon} p_t(x)dx$$

for $t = 0, 1, 2, \ldots, T$.

Plot z_t against T, setting $\epsilon = 0.1$ and $T = 600$.

Your figure should show error erratically declining something like this

34.6.3 Exercise 3

As discussed above, if the shock sequence $\{w_t\}$ is not degenerate, then it is not in general possible to predict x_t without error at time $t-1$ (and this would be the case even if we could observe x_{t-1}).

Let's now compare the prediction $\hat{x}_t$ made by the Kalman filter against a competitor who **is** allowed to observe x_{t-1}.

This competitor will use the conditional expectation $\mathbb{E}[x_t \,|\, x_{t-1}]$, which in this case is Ax_{t-1}.

The conditional expectation is known to be the optimal prediction method in terms of minimizing mean squared error.

(More precisely, the minimizer of $\mathbb{E}\,\|x_t - g(x_{t-1})\|^2$ with respect to g is $g^*(x_{t-1}) := \mathbb{E}[x_t \,|\, x_{t-1}]$)

Thus we are comparing the Kalman filter against a competitor who has more information (in the sense of being able to observe the latent state) and behaves optimally in terms of minimizing squared error.

Our horse race will be assessed in terms of squared error.

In particular, your task is to generate a graph plotting observations of both $\|x_t - Ax_{t-1}\|^2$ and $\|x_t - \hat{x}_t\|^2$ against t for $t = 1, \ldots, 50$.

For the parameters, set $G = I, R = 0.5I$ and $Q = 0.3I$, where I is the 2×2 identity.

Set

$$A = \begin{pmatrix} 0.5 & 0.4 \\ 0.6 & 0.3 \end{pmatrix}$$

To initialize the prior density, set

$$\Sigma_0 = \begin{pmatrix} 0.9 & 0.3 \\ 0.3 & 0.9 \end{pmatrix}$$

and $\hat{x}_0 = (8, 8)$.

Finally, set $x_0 = (0, 0)$.

You should end up with a figure similar to the following (modulo randomness)

Observe how, after an initial learning period, the Kalman filter performs quite well, even relative to the competitor who predicts optimally with knowledge of the latent state.

34.6.4 Exercise 4

Try varying the coefficient 0.3 in $Q = 0.3I$ up and down.

Observe how the diagonal values in the stationary solution Σ (see (9)) increase and decrease in line with this coefficient.

The interpretation is that more randomness in the law of motion for x_t causes more (permanent) uncertainty in prediction.

34.7 Solutions

34.7.1 Exercise 1

```
In [7]: # Parameters
        θ = 10  # Constant value of state x_t
        A, C, G, H = 1, 0, 1, 1
        ss = LinearStateSpace(A, C, G, H, mu_0=θ)
```

```
# Set prior, initialize kalman filter
x_hat_0, Σ_0 = 8, 1
kalman = Kalman(ss, x_hat_0, Σ_0)

# Draw observations of y from state space model
N = 5
x, y = ss.simulate(N)
y = y.flatten()

# Set up plot
fig, ax = plt.subplots(figsize=(10,8))
xgrid = np.linspace(θ - 5, θ + 2, 200)

for i in range(N):
    # Record the current predicted mean and variance
    m, v = [float(z) for z in (kalman.x_hat, kalman.Sigma)]
    # Plot, update filter
    ax.plot(xgrid, norm.pdf(xgrid, loc=m, scale=np.sqrt(v)),
↪label=f'$t={i}$')
    kalman.update(y[i])

ax.set_title(f'First {N} densities when $\\theta = {θ:.1f}$')
ax.legend(loc='upper left')
plt.show()
```

34.7.2 Exercise 2

In [8]:
```
ϵ = 0.1
θ = 10  # Constant value of state x_t
A, C, G, H = 1, 0, 1, 1
ss = LinearStateSpace(A, C, G, H, mu_0=θ)

x_hat_0, Σ_0 = 8, 1
kalman = Kalman(ss, x_hat_0, Σ_0)

T = 600
z = np.empty(T)
x, y = ss.simulate(T)
y = y.flatten()

for t in range(T):
   # Record the current predicted mean and variance and plot their densities
    m, v = [float(temp) for temp in (kalman.x_hat, kalman.Sigma)]

    f = lambda x: norm.pdf(x, loc=m, scale=np.sqrt(v))
    integral, error = quad(f, θ - ϵ, θ + ϵ)
    z[t] = 1 - integral

    kalman.update(y[t])

fig, ax = plt.subplots(figsize=(9, 7))
ax.set_ylim(0, 1)
ax.set_xlim(0, T)
ax.plot(range(T), z)
ax.fill_between(range(T), np.zeros(T), z, color="blue", alpha=0.2)
plt.show()
```

34.7.3 Exercise 3

In [9]:
```
# Define A, C, G, H
G = np.identity(2)
H = np.sqrt(0.5) * np.identity(2)

A = [[0.5, 0.4],
     [0.6, 0.3]]
C = np.sqrt(0.3) * np.identity(2)

# Set up state space mode, initial value x_0 set to zero
ss = LinearStateSpace(A, C, G, H, mu_0 = np.zeros(2))

# Define the prior density
Σ = [[0.9, 0.3],
     [0.3, 0.9]]
Σ = np.array(Σ)
x_hat = np.array([8, 8])

# Initialize the Kalman filter
kn = Kalman(ss, x_hat, Σ)

# Print eigenvalues of A
print("Eigenvalues of A:")
print(eigvals(A))

# Print stationary Σ
S, K = kn.stationary_values()
print("Stationary prediction error variance:")
print(S)

# Generate the plot
T = 50
x, y = ss.simulate(T)

e1 = np.empty(T-1)
e2 = np.empty(T-1)

for t in range(1, T):
    kn.update(y[:,t])
    e1[t-1] = np.sum((x[:, t] - kn.x_hat.flatten())**2)
    e2[t-1] = np.sum((x[:, t] - A @ x[:, t-1])**2)

fig, ax = plt.subplots(figsize=(9,6))
ax.plot(range(1, T), e1, 'k-', lw=2, alpha=0.6,
        label='Kalman filter error')
ax.plot(range(1, T), e2, 'g-', lw=2, alpha=0.6,
        label='Conditional expectation error')
ax.legend()
plt.show()
```

```
Eigenvalues of A:
[ 0.9+0.j -0.1+0.j]
Stationary prediction error variance:
[[0.40329108 0.1050718 ]
```

```
[0.1050718  0.41061709]]
```

Footnotes

[1] See, for example, page 93 of [25]. To get from his expressions to the ones used above, you will also need to apply the Woodbury matrix identity.

Chapter 35

Reverse Engineering a la Muth

35.1 Contents

- Friedman (1956) and Muth (1960) 35.2

Co-author: Chase Coleman

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

```
In [1]: !pip install --upgrade quantecon
```

We'll also need the following imports:

```
In [2]: 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 [120] about rational expectations.

Muth used *classical* prediction methods to reverse engineer a stochastic process that renders optimal Milton Friedman's [56] "adaptive expectations" scheme.

35.2 Friedman (1956) and Muth (1960)

Milton Friedman [56] (1956) posited that consumer's forecast their future disposable income with the adaptive expectations scheme

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

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 (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 [120] 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.

35.2.1 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 $\epsilon_{2,t}$:

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

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 (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*

$$\begin{aligned} \hat{x}_{t+1} &= \hat{x}_t + K a_t \\ y_t &= \hat{x}_t + a_t \end{aligned} \tag{3}$$

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

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

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

Later we'll stack these state-space systems (2) and (3) to display some classic findings of Muth.

But first, let's create an instance of the state-space system (2) then apply the quantecon `Kalman` class, then uses it to construct the associated "innovations representation"

In [3]:
```
# 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)
```

35.2.2 Some Useful State-Space Math

Now we want to map the time-invariant innovations representation (3) and the original state-space system (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

In [4]:
```
# Create grand state-space for y_t, a_t as observed vars -- Use
# stacking trick above
Af = np.array([[ 1,       0,         0],
               [K1, 1 - K1, K1 * σ_y],
               [ 0,       0,         0]])
Cf = np.array([[σ_x,         0],
               [  0, K1 * σ_y],
               [  0,         1]])
Gf = np.array([[1,  0, σ_y],
               [1, -1, σ_y]])

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

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

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

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

```
   Kalman gain = [[0.181]]
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.

35.2.3 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 $y_{t-1}, y_{t-2}, \ldots$

In [5]:
```
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*.

35.2.4 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

```
In [6]: fig, ax = plt.subplots()
        ax.plot(yf[0, :], label="y")
        ax.plot(xf[0, :], label="x")
        ax.legend()
        ax.set_title(r"$x$ and $y$")
        ax.set_xlabel("Time")
        plt.show()
```

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

35.2.5 Innovations

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

```
In [7]: 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()
```

35.2.6 MA and AR Representations

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

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

Then we'll plot each of them

In [8]:
```
# 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

```
In [9]: print(f'decay parameter 1 - K1 = {1 - K1}')
```

```
decay parameter 1 - K1 = [[0.819]]
```

Part VI

Dynamic Programming

Chapter 36

Shortest Paths

36.1 Contents

- Overview 36.2
- Outline of the Problem 36.3
- Finding Least-Cost Paths 36.4
- Solving for Minimum Cost-to-Go 36.5
- Exercises 36.6
- Solutions 36.7

36.2 Overview

The shortest path problem is a classic problem in mathematics and computer science with applications in

- Economics (sequential decision making, analysis of social networks, etc.)
- Operations research and transportation
- Robotics and artificial intelligence
- Telecommunication network design and routing
- etc., etc.

Variations of the methods we discuss in this lecture are used millions of times every day, in applications such as

- Google Maps
- routing packets on the internet

For us, the shortest path problem also provides a nice introduction to the logic of **dynamic programming**.

Dynamic programming is an extremely powerful optimization technique that we apply in many lectures on this site.

The only scientific library we'll need in what follows is NumPy:

```
In [1]: import numpy as np
```

36.3 Outline of the Problem

The shortest path problem is one of finding how to traverse a graph from one specified node to another at minimum cost.

Consider the following graph

We wish to travel from node (vertex) A to node G at minimum cost

- Arrows (edges) indicate the movements we can take.
- Numbers on edges indicate the cost of traveling that edge.

(Graphs such as the one above are called **weighted directed graphs**)

Possible interpretations of the graph include

- Minimum cost for supplier to reach a destination.
- Routing of packets on the internet (minimize time).
- Etc., etc.

For this simple graph, a quick scan of the edges shows that the optimal paths are

- A, C, F, G at cost 8

- A, D, F, G at cost 8

36.4 Finding Least-Cost Paths

For large graphs, we need a systematic solution.

Let $J(v)$ denote the minimum cost-to-go from node v, understood as the total cost from v if we take the best route.

Suppose that we know $J(v)$ for each node v, as shown below for the graph from the preceding example

Note that $J(G) = 0$.

The best path can now be found as follows

1. Start at node $v = A$
2. From current node v, move to any node that solves

$$\min_{w \in F_v} \{c(v, w) + J(w)\} \tag{1}$$

where

- F_v is the set of nodes that can be reached from v in one step.
- $c(v, w)$ is the cost of traveling from v to w.

Hence, if we know the function J, then finding the best path is almost trivial.

But how can we find the cost-to-go function J?

Some thought will convince you that, for every node v, the function J satisfies

$$J(v) = \min_{w \in F_v} \{c(v, w) + J(w)\} \tag{2}$$

This is known as the *Bellman equation*, after the mathematician Richard Bellman.

The Bellman equation can be thought of as a restriction that J must satisfy.

What we want to do now is use this restriction to compute J.

36.5 Solving for Minimum Cost-to-Go

Let's look at an algorithm for computing J and then think about how to implement it.

36.5.1 The Algorithm

The standard algorithm for finding J is to start an initial guess and then iterate.

This is a standard approach to solving nonlinear equations, often called the method of **successive approximations**.

Our initial guess will be

$$J_0(v) = 0 \text{ for all } v \tag{3}$$

Now

1. Set $n = 0$
2. Set $J_{n+1}(v) = \min_{w \in F_v} \{c(v, w) + J_n(w)\}$ for all v
3. If J_{n+1} and J_n are not equal then increment n, go to 2

This sequence converges to J.

Although we omit the proof, we'll prove similar claims in our other lectures on dynamic programming.

36.5.2 Implementation

Having an algorithm is a good start, but we also need to think about how to implement it on a computer.

First, for the cost function c, we'll implement it as a matrix Q, where a typical element is

$$Q(v, w) = \begin{cases} c(v, w) \text{ if } w \in F_v \\ +\infty \text{ otherwise} \end{cases}$$

In this context Q is usually called the **distance matrix**.

We're also numbering the nodes now, with $A = 0$, so, for example

$$Q(1, 2) = \text{ the cost of traveling from B to C}$$

For example, for the simple graph above, we set

```
In [2]: from numpy import inf

        Q = np.array([[inf, 1,   5,   3,   inf, inf, inf],
                      [inf, inf, inf, 9,   6,   inf, inf],
                      [inf, inf, inf, inf, inf, 2,   inf],
                      [inf, inf, inf, inf, inf, 4,   8],
                      [inf, inf, inf, inf, inf, inf, 4],
                      [inf, inf, inf, inf, inf, inf, 1],
                      [inf, inf, inf, inf, inf, inf, 0]])
```

Notice that the cost of staying still (on the principle diagonal) is set to

- np.inf for non-destination nodes — moving on is required.
- 0 for the destination node — here is where we stop.

For the sequence of approximations $\{J_n\}$ of the cost-to-go functions, we can use NumPy arrays.

Let's try with this example and see how we go:

In [3]:
```
nodes = range(7)                                # Nodes = 0, 1, ..., 6
J = np.zeros_like(nodes, dtype=np.int)          # Initial guess
next_J = np.empty_like(nodes, dtype=np.int)     # Stores updated guess

max_iter = 500
i = 0

while i < max_iter:
    for v in nodes:
        # minimize Q[v, w] + J[w] over all choices of w
        lowest_cost = inf
        for w in nodes:
            cost = Q[v, w] + J[w]
            if cost < lowest_cost:
                lowest_cost = cost
        next_J[v] = lowest_cost
    if np.equal(next_J, J).all():
        break
    else:
        J[:] = next_J   # Copy contents of next_J to J
        i += 1

print("The cost-to-go function is", J)
```

```
The cost-to-go function is [ 8 10  3  5  4  1  0]
```

This matches with the numbers we obtained by inspection above.

But, importantly, we now have a methodology for tackling large graphs.

36.6 Exercises

36.6.1 Exercise 1

The text below describes a weighted directed graph.

The line `node0, node1 0.04, node8 11.11, node14 72.21` means that from node0 we can go to

- node1 at cost 0.04
- node8 at cost 11.11
- node14 at cost 72.21

No other nodes can be reached directly from node0.

Other lines have a similar interpretation.

Your task is to use the algorithm given above to find the optimal path and its cost.

Note: You will be dealing with floating point numbers now, rather than integers, so consider replacing `np.equal()` with `np.allclose()`.

In [4]:
```
%%file graph.txt
node0, node1 0.04, node8 11.11, node14 72.21
node1, node46 1247.25, node6 20.59, node13 64.94
node2, node66 54.18, node31 166.80, node45 1561.45
node3, node20 133.65, node6 2.06, node11 42.43
node4, node75 3706.67, node5 0.73, node7 1.02
node5, node45 1382.97, node7 3.33, node11 34.54
node6, node31 63.17, node9 0.72, node10 13.10
node7, node50 478.14, node9 3.15, node10 5.85
node8, node69 577.91, node11 7.45, node12 3.18
node9, node70 2454.28, node13 4.42, node20 16.53
node10, node89 5352.79, node12 1.87, node16 25.16
node11, node94 4961.32, node18 37.55, node20 65.08
node12, node84 3914.62, node24 34.32, node28 170.04
node13, node60 2135.95, node38 236.33, node40 475.33
node14, node67 1878.96, node16 2.70, node24 38.65
node15, node91 3597.11, node17 1.01, node18 2.57
node16, node36 392.92, node19 3.49, node38 278.71
node17, node76 783.29, node22 24.78, node23 26.45
node18, node91 3363.17, node23 16.23, node28 55.84
node19, node26 20.09, node20 0.24, node28 70.54
node20, node98 3523.33, node24 9.81, node33 145.80
node21, node56 626.04, node28 36.65, node31 27.06
node22, node72 1447.22, node39 136.32, node40 124.22
node23, node52 336.73, node26 2.66, node33 22.37
node24, node66 875.19, node26 1.80, node28 14.25
node25, node70 1343.63, node32 36.58, node35 45.55
node26, node47 135.78, node27 0.01, node42 122.00
node27, node65 480.55, node35 48.10, node43 246.24
node28, node82 2538.18, node34 21.79, node36 15.52
node29, node64 635.52, node32 4.22, node33 12.61
node30, node98 2616.03, node33 5.61, node35 13.95
node31, node98 3350.98, node36 20.44, node44 125.88
node32, node97 2613.92, node34 3.33, node35 1.46
node33, node81 1854.73, node41 3.23, node47 111.54
node34, node73 1075.38, node42 51.52, node48 129.45
node35, node52 17.57, node41 2.09, node50 78.81
node36, node71 1171.60, node54 101.08, node57 260.46
node37, node75 269.97, node38 0.36, node46 80.49
node38, node93 2767.85, node40 1.79, node42 8.78
node39, node50 39.88, node40 0.95, node41 1.34
node40, node75 548.68, node47 28.57, node54 53.46
node41, node53 18.23, node46 0.28, node54 162.24
node42, node59 141.86, node47 10.08, node72 437.49
node43, node98 2984.83, node54 95.06, node60 116.23
node44, node91 807.39, node46 1.56, node47 2.14
node45, node58 79.93, node47 3.68, node49 15.51
node46, node52 22.68, node57 27.50, node67 65.48
node47, node50 2.82, node56 49.31, node61 172.64
node48, node99 2564.12, node59 34.52, node60 66.44
node49, node78 53.79, node50 0.51, node56 10.89
node50, node85 251.76, node53 1.38, node55 20.10
node51, node98 2110.67, node59 23.67, node60 73.79
node52, node94 1471.80, node64 102.41, node66 123.03
node53, node72 22.85, node56 4.33, node67 88.35
```

```
node54, node88 967.59, node59 24.30, node73 238.61
node55, node84 86.09, node57 2.13, node64 60.80
node56, node76 197.03, node57 0.02, node61 11.06
node57, node86 701.09, node58 0.46, node60 7.01
node58, node83 556.70, node64 29.85, node65 34.32
node59, node90 820.66, node60 0.72, node71 0.67
node60, node76 48.03, node65 4.76, node67 1.63
node61, node98 1057.59, node63 0.95, node64 4.88
node62, node91 132.23, node64 2.94, node76 38.43
node63, node66 4.43, node72 70.08, node75 56.34
node64, node80 47.73, node65 0.30, node76 11.98
node65, node94 594.93, node66 0.64, node73 33.23
node66, node98 395.63, node68 2.66, node73 37.53
node67, node82 153.53, node68 0.09, node70 0.98
node68, node94 232.10, node70 3.35, node71 1.66
node69, node99 247.80, node70 0.06, node73 8.99
node70, node76 27.18, node72 1.50, node73 8.37
node71, node89 104.50, node74 8.86, node91 284.64
node72, node76 15.32, node84 102.77, node92 133.06
node73, node83 52.22, node76 1.40, node90 243.00
node74, node81 1.07, node76 0.52, node78 8.08
node75, node92 68.53, node76 0.81, node77 1.19
node76, node85 13.18, node77 0.45, node78 2.36
node77, node80 8.94, node78 0.98, node86 64.32
node78, node98 355.90, node81 2.59
node79, node81 0.09, node85 1.45, node91 22.35
node80, node92 121.87, node88 28.78, node98 264.34
node81, node94 99.78, node89 39.52, node92 99.89
node82, node91 47.44, node88 28.05, node93 11.99
node83, node94 114.95, node86 8.75, node88 5.78
node84, node89 19.14, node94 30.41, node98 121.05
node85, node97 94.51, node87 2.66, node89 4.90
node86, node97 85.09
node87, node88 0.21, node91 11.14, node92 21.23
node88, node93 1.31, node91 6.83, node98 6.12
node89, node97 36.97, node99 82.12
node90, node96 23.53, node94 10.47, node99 50.99
node91, node97 22.17
node92, node96 10.83, node97 11.24, node99 34.68
node93, node94 0.19, node97 6.71, node99 32.77
node94, node98 5.91, node96 2.03
node95, node98 6.17, node99 0.27
node96, node98 3.32, node97 0.43, node99 5.87
node97, node98 0.30
node98, node99 0.33
node99,
```

```
Overwriting graph.txt
```

36.7 Solutions

36.7.1 Exercise 1

First let's write a function that reads in the graph data above and builds a distance matrix.

In [5]:
```
num_nodes = 100
destination_node = 99

def map_graph_to_distance_matrix(in_file):

    # First let's set of the distance matrix Q with inf everywhere
    Q = np.ones((num_nodes, num_nodes))
    Q = Q * np.inf

    # Now we read in the data and modify Q
    infile = open(in_file)
    for line in infile:
        elements = line.split(',')
        node = elements.pop(0)
        node = int(node[4:])    # convert node description to integer
        if node != destination_node:
            for element in elements:
                destination, cost = element.split()
                destination = int(destination[4:])
                Q[node, destination] = float(cost)
        Q[destination_node, destination_node] = 0

    infile.close()
    return Q
```

In addition, let's write

1. a "Bellman operator" function that takes a distance matrix and current guess of J and returns an updated guess of J, and
2. a function that takes a distance matrix and returns a cost-to-go function.

We'll use the algorithm described above.

The minimization step is vectorized to make it faster.

In [6]:
```
def bellman(J, Q):
    num_nodes = Q.shape[0]
    next_J = np.empty_like(J)
    for v in range(num_nodes):
        next_J[v] = np.min(Q[v, :] + J)
    return next_J


def compute_cost_to_go(Q):
    J = np.zeros(num_nodes)      # Initial guess
    next_J = np.empty(num_nodes)  # Stores updated guess
    max_iter = 500
    i = 0

    while i < max_iter:
        next_J = bellman(J, Q)
        if np.allclose(next_J, J):
            break
        else:
            J[:] = next_J   # Copy contents of next_J to J
```

```
            i += 1

    return(J)
```

We used np.allclose() rather than testing exact equality because we are dealing with floating point numbers now.

Finally, here's a function that uses the cost-to-go function to obtain the optimal path (and its cost).

```
In [7]: def print_best_path(J, Q):
            sum_costs = 0
            current_node = 0
            while current_node != destination_node:
                print(current_node)
                # Move to the next node and increment costs
                next_node = np.argmin(Q[current_node, :] + J)
                sum_costs += Q[current_node, next_node]
                current_node = next_node

            print(destination_node)
            print('Cost: ', sum_costs)
```

Okay, now we have the necessary functions, let's call them to do the job we were assigned.

```
In [8]: Q = map_graph_to_distance_matrix('graph.txt')
        J = compute_cost_to_go(Q)
        print_best_path(J, Q)
```

```
    0
8
11
18
23
33
41
53
56
57
60
67
70
73
76
85
87
88
93
94
96
97
98
99
Cost:  160.55000000000007
```

Chapter 37

Job Search I: The McCall Search Model

37.1 Contents

- Overview 37.2
- The McCall Model 37.3
- Computing the Optimal Policy: Take 1 37.4
- Computing the Optimal Policy: Take 2 37.5
- Exercises 37.6
- Solutions 37.7

> "Questioning a McCall worker is like having a conversation with an out-of-work friend: 'Maybe you are setting your sights too high', or 'Why did you quit your old job before you had a new one lined up?' This is real social science: an attempt to model, to understand, human behavior by visualizing the situation people find themselves in, the options they face and the pros and cons as they themselves see them." – Robert E. Lucas, Jr.

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

```
In [1]: !pip install --upgrade quantecon
```

37.2 Overview

The McCall search model [116] helped transform economists' way of thinking about labor markets.

To clarify vague notions such as "involuntary" unemployment, McCall modeled the decision problem of unemployed agents directly, in terms of factors such as

- current and likely future wages
- impatience
- unemployment compensation

To solve the decision problem he used dynamic programming.

Here we set up McCall's model and adopt the same solution method.

As we'll see, McCall's model is not only interesting in its own right but also an excellent vehicle for learning dynamic programming.

Let's start with some imports:

In [2]:
```
import numpy as np
from numba import jit, jitclass, float64
import matplotlib.pyplot as plt
%matplotlib inline
import quantecon as qe
from quantecon.distributions import BetaBinomial
```

37.3 The McCall Model

An unemployed agent receives in each period a job offer at wage w_t.

The wage offer is a nonnegative function of some underlying state:

$$w_t = w(s_t) \quad \text{where} \quad s_t \in \mathbb{S}$$

Here you should think of state process $\{s_t\}$ as some underlying, unspecified random factor that impacts on wages.

(Introducing an exogenous stochastic state process is a standard way for economists to inject randomness into their models.)

In this lecture, we adopt the following simple environment:

- $\{s_t\}$ is IID, with $q(s)$ being the probability of observing state s in $\mathbb{S}$ at each point in time, and
- the agent observes s_t at the start of t and hence knows $w_t = w(s_t)$,
- the set $\mathbb{S}$ is finite.

(In later lectures, we will relax all of these assumptions.)

At time t, our agent has two choices:

1. Accept the offer and work permanently at constant wage w_t.
2. Reject the offer, receive unemployment compensation c, and reconsider next period.

The agent is infinitely lived and aims to maximize the expected discounted sum of earnings

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t$$

The constant β lies in $(0, 1)$ and is called a **discount factor**.

The smaller is β, the more the agent discounts future utility relative to current utility.

The variable y_t is income, equal to

- his/her wage w_t when employed

- unemployment compensation c when unemployed

The agent is assumed to know that $\{s_t\}$ is IID with common distribution q and can use this when computing expectations.

37.3.1 A Trade-Off

The worker faces a trade-off:

- Waiting too long for a good offer is costly, since the future is discounted.
- Accepting too early is costly, since better offers might arrive in the future.

To decide optimally in the face of this trade-off, we use dynamic programming.

Dynamic programming can be thought of as a two-step procedure that

1. first assigns values to "states" and
2. then deduces optimal actions given those values

We'll go through these steps in turn.

37.3.2 The Value Function

In order to optimally trade-off current and future rewards, we need to think about two things:

1. the current payoffs we get from different choices
2. the different states that those choices will lead to in next period (in this case, either employment or unemployment)

To weigh these two aspects of the decision problem, we need to assign *values* to states.

To this end, let $v^*(s)$ be the total lifetime *value* accruing to an unemployed worker who enters the current period unemployed when the state is $s \in \mathbb{S}$.

In particular, the agent has wage offer $w(s)$ in hand.

More precisely, $v^*(s)$ denotes the value of the objective function (1) when an agent in this situation makes *optimal* decisions now and at all future points in time.

Of course $v^*(s)$ is not trivial to calculate because we don't yet know what decisions are optimal and what aren't!

But think of v^* as a function that assigns to each possible state s the maximal lifetime value that can be obtained with that offer in hand.

A crucial observation is that this function v^* must satisfy the recursion

$$v^*(s) = \max \left\{ \frac{w(s)}{1-\beta}, \, c + \beta \sum_{s' \in \mathbb{S}} v^*(s') q(s') \right\} \tag{1}$$

for every possible s in $\mathbb{S}$.

This important equation is a version of the **Bellman equation**, which is ubiquitous in economic dynamics and other fields involving planning over time.

The intuition behind it is as follows:

- the first term inside the max operation is the lifetime payoff from accepting current offer, since

$$\frac{w(s)}{1-\beta} = w(s) + \beta w(s) + \beta^2 w(s) + \cdots$$

- the second term inside the max operation is the **continuation value**, which is the lifetime payoff from rejecting the current offer and then behaving optimally in all subsequent periods

If we optimize and pick the best of these two options, we obtain maximal lifetime value from today, given current state s.

But this is precisely $v^*(s)$, which is the l.h.s. of (1).

37.3.3 The Optimal Policy

Suppose for now that we are able to solve (1) for the unknown function v^*.

Once we have this function in hand we can behave optimally (i.e., make the right choice between accept and reject).

All we have to do is select the maximal choice on the r.h.s. of (1).

The optimal action is best thought of as a **policy**, which is, in general, a map from states to actions.

Given *any* s, we can read off the corresponding best choice (accept or reject) by picking the max on the r.h.s. of (1).

Thus, we have a map from $\mathbb{R}$ to $\{0, 1\}$, with 1 meaning accept and 0 meaning reject.

We can write the policy as follows

$$\sigma(s) := \mathbf{1}\left\{\frac{w(s)}{1-\beta} \geq c + \beta \sum_{s' \in \mathbb{S}} v^*(s')q(s')\right\}$$

Here $\mathbf{1}\{P\} = 1$ if statement P is true and equals 0 otherwise.

We can also write this as

$$\sigma(s) := \mathbf{1}\{w(s) \geq \bar{w}\}$$

where

$$\bar{w} := (1-\beta)\left\{c + \beta \sum_{s'} v^*(s')q(s')\right\} \tag{2}$$

Here $\bar{w}$ (called the *reservation wage*) is a constant depending on β, c and the wage distribution.

The agent should accept if and only if the current wage offer exceeds the reservation wage.

In view of (2), we can compute this reservation wage if we can compute the value function.

37.4 Computing the Optimal Policy: Take 1

To put the above ideas into action, we need to compute the value function at each possible state $s \in \mathbb{S}$.

Let's suppose that $\mathbb{S} = \{1, \dots, n\}$.

The value function is then represented by the vector $v^* = (v^*(i))_{i=1}^n$.

In view of (1), this vector satisfies the nonlinear system of equations

$$v^*(i) = \max \left\{ \frac{w(i)}{1-\beta}, c + \beta \sum_{1 \leq j \leq n} v^*(j) q(j) \right\} \quad \text{for } i = 1, \dots, n \tag{3}$$

37.4.1 The Algorithm

To compute this vector, we use successive approximations:

Step 1: pick an arbitrary initial guess $v \in \mathbb{R}^n$.

Step 2: compute a new vector $v' \in \mathbb{R}^n$ via

$$v'(i) = \max \left\{ \frac{w(i)}{1-\beta}, c + \beta \sum_{1 \leq j \leq n} v(j) q(j) \right\} \quad \text{for } i = 1, \dots, n \tag{4}$$

Step 3: calculate a measure of the deviation between v and v', such as $\max_i |v(i) - v'(i)|$.

Step 4: if the deviation is larger than some fixed tolerance, set $v = v'$ and go to step 2, else continue.

Step 5: return v.

Let $\{v_k\}$ denote the sequence genererated by this algorithm.

This sequence converges to the solution to (3) as $k \to \infty$, which is the value function v^*.

37.4.2 The Fixed Point Theory

What's the mathematics behind these ideas?

First, one defines a mapping T from $\mathbb{R}^n$ to itself via

$$(Tv)(i) = \max \left\{ \frac{w(i)}{1-\beta}, c + \beta \sum_{1 \leq j \leq n} v(j) q(j) \right\} \quad \text{for } i = 1, \dots, n \tag{5}$$

(A new vector Tv is obtained from given vector v by evaluating the r.h.s. at each i)

The element v_k in the sequence $\{v_k\}$ of successive approximations corresponds to $T^k v$.

- This is T applied k times, starting at the initial guess v

One can show that the conditions of the Banach fixed point theorem are satisfied by T on $\mathbb{R}^n$.

One implication is that T has a unique fixed point in $\mathbb{R}^n$.

- That is, a unique vector $\bar{v}$ such that $T\bar{v} = \bar{v}$.

Moreover, it's immediate from the definition of T that this fixed point is v^*.

A second implication of the Banach contraction mapping theorem is that $\{T^k v\}$ converges to the fixed point v^* regardless of v.

37.4.3 Implementation

Our default for q, the distribution of the state process, will be Beta-binomial.

```
In [3]: n, a, b = 50, 200, 100                          # default parameters
        q_default = BetaBinomial(n, a, b).pdf()         # default choice of q
```

Our default set of values for wages will be

```
In [4]: w_min, w_max = 10, 60
        w_default = np.linspace(w_min, w_max, n+1)
```

Here's a plot of the probabilities of different wage outcomes:

```
In [5]: fig, ax = plt.subplots()
        ax.plot(w_default, q_default, '-o', label='$q(w(i))$')
        ax.set_xlabel('wages')
        ax.set_ylabel('probabilities')

        plt.show()
```

We are going to use Numba to accelerate our code.

- See, in particular, the discussion of `@jitclass` in our lecture on Numba.

The following helps Numba by providing some type

```
In [6]: mccall_data = [
            ('c', float64),       # unemployment compensation
            ('β', float64),       # discount factor
            ('w', float64[:]),    # array of wage values, w[i] = wage at state i
            ('q', float64[:])     # array of probabilities
        ]
```

Here's a class that stores the data and computes the value on the right hand side of the Bellman equation (4).

Default parameter values are embedded in the class.

```
In [7]: @jitclass(mccall_data)
        class McCallModel:

            def __init__(self, c=25, β=0.99, w=w_default, q=q_default):

                self.c, self.β = c, β
                self.w, self.q = w_default, q_default

            def bellman(self, i, v):
                """
                The r.h.s. of the Bellman equation at state i.
                """
                # Simplify names
                c, β, w, q = self.c, self.β, self.w, self.q
                # Evaluate right hand side of Bellman equation
                max_value = max(w[i] / (1 - β), c + β * np.sum(v * q))
                return(max_value)
```

Based on these defaults, let's try plotting the first few approximate value functions in the sequence $\{T^k v\}$.

We will start from guess v given by $v(i) = w(i)/(1-\beta)$, which is the value of accepting at every given wage.

Here's a function to implement this:

```
In [8]: def plot_value_function_seq(mcm, ax, num_plots=6):
            """
            Plot a sequence of value functions.

                * mcm is an instance of McCallModel
                * ax is an axes object that implements a plot method.

            """

            n = len(mcm.w)
            v = mcm.w / (1 - mcm.β)
            v_next = np.empty_like(v)
```

```
    for i in range(num_plots):
        ax.plot(mcm.w, v, '-', alpha=0.4, label=f"iterate {i}")
        # Update guess
        for i in range(n):
            v_next[i] = mcm.bellman(i, v)
        v[:] = v_next  # copy contents into v

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

Now let's create an instance of `McCallModel` and call the function:

```
In [9]: mcm = McCallModel()

        fig, ax = plt.subplots()
        ax.set_xlabel('wage')
        ax.set_ylabel('value')
        plot_value_function_seq(mcm, ax)
        plt.show()
```

You can see that convergence is occuring: successive iterates are getting closer together.

Here's a more serious iteration effort to compute the limit, which continues until measured deviation between successive iterates is below tol.

Once we obtain a good approximation to the limit, we will use it to calculate the reservation wage.

We'll be using JIT compilation via Numba to turbocharge our loops.

```
In [10]: @jit(nopython=True)
         def compute_reservation_wage(mcm,
                                      max_iter=500,
```

```
                                tol=1e-6):

    # Simplify names
    c, β, w, q = mcm.c, mcm.β, mcm.w, mcm.q

    # == First compute the value function == #

    n = len(w)
    v = w / (1 - β)          # initial guess
    v_next = np.empty_like(v)
    i = 0
    error = tol + 1
    while i < max_iter and error > tol:

        for i in range(n):
            v_next[i] = mcm.bellman(i, v)

        error = np.max(np.abs(v_next - v))
        i += 1

        v[:] = v_next  # copy contents into v

    # == Now compute the reservation wage == #

    return (1 - β) * (c + β * np.sum(v * q))
```

The next line computes the reservation wage at the default parameters

```
In [11]: compute_reservation_wage(mcm)
```

```
Out[11]: 47.316499710024964
```

37.4.4 Comparative Statics

Now we know how to compute the reservation wage, let's see how it varies with parameters.

In particular, let's look at what happens when we change β and c.

```
In [12]: grid_size = 25
         R = np.empty((grid_size, grid_size))

         c_vals = np.linspace(10.0, 30.0, grid_size)
         β_vals = np.linspace(0.9, 0.99, grid_size)

         for i, c in enumerate(c_vals):
             for j, β in enumerate(β_vals):
                 mcm = McCallModel(c=c, β=β)
                 R[i, j] = compute_reservation_wage(mcm)
```

```
In [13]: fig, ax = plt.subplots()

         cs1 = ax.contourf(c_vals, β_vals, R.T, alpha=0.75)
         ctr1 = ax.contour(c_vals, β_vals, R.T)

         plt.clabel(ctr1, inline=1, fontsize=13)
```

```
plt.colorbar(cs1, ax=ax)


ax.set_title("reservation wage")
ax.set_xlabel("$c$", fontsize=16)
ax.set_ylabel("$β$", fontsize=16)

ax.ticklabel_format(useOffset=False)

plt.show()
```

As expected, the reservation wage increases both with patience and with unemployment compensation.

37.5 Computing the Optimal Policy: Take 2

The approach to dynamic programming just described is very standard and broadly applicable.

For this particular problem, there's also an easier way, which circumvents the need to compute the value function.

Let h denote the continuation value:

$$h = c + \beta \sum_{s'} v^*(s') q(s') \tag{6}$$

The Bellman equation can now be written as

$$v^*(s') = \max\left\{\frac{w(s')}{1-\beta}, h\right\}$$

Substituting this last equation into (6) gives

$$h = c + \beta \sum_{s' \in \mathbb{S}} \max\left\{\frac{w(s')}{1-\beta}, h\right\} q(s') \tag{7}$$

This is a nonlinear equation that we can solve for h.

As before, we will use successive approximations:

Step 1: pick an initial guess h.

Step 2: compute the update h' via

$$h' = c + \beta \sum_{s' \in \mathbb{S}} \max\left\{\frac{w(s')}{1-\beta}, h\right\} q(s') \tag{8}$$

Step 3: calculate the deviation $|h - h'|$.

Step 4: if the deviation is larger than some fixed tolerance, set $h = h'$ and go to step 2, else return h.

Once again, one can use the Banach contraction mapping theorem to show that this process always converges.

The big difference here, however, is that we're iterating on a single number, rather than an n-vector.

Here's an implementation:

In [14]:
```
@jit(nopython=True)
def compute_reservation_wage_two(mcm,
                                 max_iter=500,
                                 tol=1e-5):

    # Simplify names
    c, β, w, q = mcm.c, mcm.β, mcm.w, mcm.q

    # == First compute h == #

    h = np.sum(w * q) / (1 - β)
    i = 0
    error = tol + 1
    while i < max_iter and error > tol:

        s = np.maximum(w / (1 - β), h)
        h_next = c + β * np.sum(s * q)

        error = np.abs(h_next - h)
        i += 1

        h = h_next

    # == Now compute the reservation wage == #
```

```
    return (1 - β) * h
```

You can use this code to solve the exercise below.

37.6 Exercises

37.6.1 Exercise 1

Compute the average duration of unemployment when $\beta = 0.99$ and c takes the following values

```
c_vals = np.linspace(10, 40, 25)
```

That is, start the agent off as unemployed, compute their reservation wage given the parameters, and then simulate to see how long it takes to accept.

Repeat a large number of times and take the average.

Plot mean unemployment duration as a function of c in `c_vals`.

37.6.2 Exercise 2

The purpose of this exercise is to show how to replace the discrete wage offer distribution used above with a continuous distribution.

This is a significant topic because many convenient distributions are continuous (i.e., have a density).

Fortunately, the theory changes little in our simple model.

Recall that h in (6) denotes the value of not accepting a job in this period but then behaving optimally in all subsequent periods:

To shift to a continuous offer distribution, we can replace (6) by

$$h = c + \beta \int v^*(s')q(s')ds'. \tag{9}$$

Equation (7) becomes

$$h = c + \beta \int \max\left\{\frac{w(s')}{1-\beta}, h\right\} q(s')ds' \tag{10}$$

The aim is to solve this nonlinear equation by iteration, and from it obtain the reservation wage.

Try to carry this out, setting

- the state sequence $\{s_t\}$ to be IID and standard normal and
- the wage function to be $w(s) = \exp(\mu + \sigma s)$.

You will need to implement a new version of the `McCallModel` class that assumes a lognormal wage distribution.

Calculate the integral by Monte Carlo, by averaging over a large number of wage draws.

For default parameters, use `c=25, β=0.99, σ=0.5, μ=2.5`.

Once your code is working, investigate how the reservation wage changes with c and β.

37.7 Solutions

37.7.1 Exercise 1

Here's one solution

```
In [15]: cdf = np.cumsum(q_default)

         @jit(nopython=True)
         def compute_stopping_time(w_bar, seed=1234):

             np.random.seed(seed)
             t = 1
             while True:
                 # Generate a wage draw
                 w = w_default[qe.random.draw(cdf)]
                 # Stop when the draw is above the reservation wage
                 if w >= w_bar:
                     stopping_time = t
                     break
                 else:
                     t += 1
             return stopping_time

         @jit(nopython=True)
         def compute_mean_stopping_time(w_bar, num_reps=100000):
             obs = np.empty(num_reps)
             for i in range(num_reps):
                 obs[i] = compute_stopping_time(w_bar, seed=i)
             return obs.mean()

         c_vals = np.linspace(10, 40, 25)
         stop_times = np.empty_like(c_vals)
         for i, c in enumerate(c_vals):
             mcm = McCallModel(c=c)
             w_bar = compute_reservation_wage_two(mcm)
             stop_times[i] = compute_mean_stopping_time(w_bar)

         fig, ax = plt.subplots()

         ax.plot(c_vals, stop_times, label="mean unemployment duration")
         ax.set(xlabel="unemployment compensation", ylabel="months")
         ax.legend()

         plt.show()
```

37.7.2 Exercise 2

```
In [16]: mccall_data_continuous = [
             ('c', float64),           # unemployment compensation
             ('β', float64),           # discount factor
             ('σ', float64),           # scale parameter in lognormal distribution
            ('μ', float64),          # location parameter in lognormal distribution
             ('w_draws', float64[:])  # draws of wages for Monte Carlo
         ]

         @jitclass(mccall_data_continuous)
         class McCallModelContinuous:

             def __init__(self, c=25, β=0.99, σ=0.5, μ=2.5, mc_size=1000):

                 self.c, self.β, self.σ, self.μ = c, β, σ, μ

                 # Draw and store shocks
                 np.random.seed(1234)
                 s = np.random.randn(mc_size)
                 self.w_draws = np.exp(μ+ σ * s)


         @jit(nopython=True)
         def compute_reservation_wage_continuous(mcmc, max_iter=500, tol=1e-5):

             c, β, σ, μ, w_draws = mcmc.c, mcmc.β, mcmc.σ, mcmc.μ, mcmc.w_draws

             h = np.mean(w_draws) / (1 - β)  # initial guess
             i = 0
             error = tol + 1
             while i < max_iter and error > tol:
```

```
        integral = np.mean(np.maximum(w_draws / (1 - β), h))
        h_next = c + β * integral

        error = np.abs(h_next - h)
        i += 1

        h = h_next

    # == Now compute the reservation wage == #

    return (1 - β) * h
```

Now we investigate how the reservation wage changes with c and β.

We will do this using a contour plot.

```
In [17]: grid_size = 25
         R = np.empty((grid_size, grid_size))

         c_vals = np.linspace(10.0, 30.0, grid_size)
         β_vals = np.linspace(0.9, 0.99, grid_size)

         for i, c in enumerate(c_vals):
             for j, β in enumerate(β_vals):
                 mcmc = McCallModelContinuous(c=c, β=β)
                 R[i, j] = compute_reservation_wage_continuous(mcmc)
```

```
In [18]: fig, ax = plt.subplots()

         cs1 = ax.contourf(c_vals, β_vals, R.T, alpha=0.75)
         ctr1 = ax.contour(c_vals, β_vals, R.T)

         plt.clabel(ctr1, inline=1, fontsize=13)
         plt.colorbar(cs1, ax=ax)


         ax.set_title("reservation wage")
         ax.set_xlabel("$c$", fontsize=16)
         ax.set_ylabel("$β$", fontsize=16)

         ax.ticklabel_format(useOffset=False)

         plt.show()
```

reservation wage
0.99
0.98
0.97
0.96
0.95
0.94
0.93
0.92
0.91
0.90
β
10.0
12.5
15.0
17.5
20.0
22.5
25.0
27.5
30.0
c
32.500
30.000
27.500
25.000
22.500
37.5
35.0
32.5
30.0
27.5
25.0
22.5
20.0
17.5

Chapter 38

Job Search II: Search and Separation

38.1 Contents

- Overview 38.2
- The Model 38.3
- Solving the Model 38.4
- Implementation 38.5
- Impact of Parameters 38.6
- Exercises 38.7
- Solutions 38.8

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

```
In [1]: !pip install --upgrade quantecon
```

38.2 Overview

Previously we looked at the McCall job search model [116] as a way of understanding unemployment and worker decisions.

One unrealistic feature of the model is that every job is permanent.

In this lecture, we extend the McCall model by introducing job separation.

Once separation enters the picture, the agent comes to view

- the loss of a job as a capital loss, and
- a spell of unemployment as an *investment* in searching for an acceptable job

The other minor addition is that a utility function will be included to make worker preferences slightly more sophisticated.

We'll need the following imports

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

```
from numba import njit, jitclass, float64
from quantecon.distributions import BetaBinomial
```

38.3 The Model

The model is similar to the baseline McCall job search model.

It concerns the life of an infinitely lived worker and

- the opportunities he or she (let's say he to save one character) has to work at different wages
- exogenous events that destroy his current job
- his decision making process while unemployed

The worker can be in one of two states: employed or unemployed.

He wants to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(y_t) \tag{1}$$

At this stage the only difference from the baseline model is that we've added some flexibility to preferences by introducing a utility function u.

It satisfies $u' > 0$ and $u'' < 0$.

38.3.1 The Wage Process

For now we will drop the separation of state process and wage process that we maintained for the baseline model.

In particular, we simply suppose that wage offers $\{w_t\}$ are IID with common distribution q.

The set of possible wage values is denoted by $\mathbb{W}$.

(Later we will go back to having a separate state process $\{s_t\}$ driving random outcomes, since this formulation is usually convenient in more sophisticated models.)

38.3.2 Timing and Decisions

At the start of each period, the agent can be either

- unemployed or
- employed at some existing wage level w_e.

At the start of a given period, the current wage offer w_t is observed.

If currently *employed*, the worker

1. receives utility $u(w_e)$ and
2. is fired with some (small) probability α.

If currently *unemployed*, the worker either accepts or rejects the current offer w_t.

If he accepts, then he begins work immediately at wage w_t.

If he rejects, then he receives unemployment compensation c.

The process then repeats.

(Note: we do not allow for job search while employed—this topic is taken up in a later lecture)

38.4 Solving the Model

We drop time subscripts in what follows and primes denote next period values.

Let

- $v(w_e)$ be total lifetime value accruing to a worker who enters the current period *employed* with existing wage w_e
- $h(w)$ be total lifetime value accruing to a worker who who enters the current period *unemployed* and receives wage offer w.

Here *value* means the value of the objective function (1) when the worker makes optimal decisions at all future points in time.

Our first aim is to obtain these functions.

38.4.1 The Bellman Equations

Suppose for now that the worker can calculate the functions v and h and use them in his decision making.

Then v and h should satisfy

$$v(w_e) = u(w_e) + \beta \left[(1-\alpha) v(w_e) + \alpha \sum_{w' \in \mathbb{W}} h(w') q(w') \right] \tag{2}$$

and

$$h(w) = \max \left\{ v(w),\, u(c) + \beta \sum_{w' \in \mathbb{W}} h(w') q(w') \right\} \tag{3}$$

Equation (2) expresses the value of being employed at wage w_e in terms of

- current reward $u(w_e)$ plus
- discounted expected reward tomorrow, given the α probability of being fired

Equation (3) expresses the value of being unemployed with offer w in hand as a maximum over the value of two options: accept or reject the current offer.

Accepting transitions the worker to employment and hence yields reward $v(w)$.

Rejecting leads to unemployment compensation and unemployment tomorrow.

Equations (2) and (3) are the Bellman equations for this model.

They provide enough information to solve for both v and h.

38.4.2 A Simplifying Transformation

Rather than jumping straight into solving these equations, let's see if we can simplify them somewhat.

(This process will be analogous to our second pass at the plain vanilla McCall model, where we simplified the Bellman equation.)

First, let

$$d := \sum_{w' \in \mathbb{W}} h(w')q(w') \tag{4}$$

be the expected value of unemployment tomorrow.

We can now write (3) as

$$h(w) = \max\{v(w),\, u(c) + \beta d\}$$

or, shifting time forward one period

$$\sum_{w' \in \mathbb{W}} h(w')q(w') = \sum_{w' \in \mathbb{W}} \max\{v(w'),\, u(c) + \beta d\}\, q(w')$$

Using (4) again now gives

$$d = \sum_{w' \in \mathbb{W}} \max\{v(w'),\, u(c) + \beta d\}\, q(w') \tag{5}$$

Finally, (2) can now be rewritten as

$$v(w) = u(w) + \beta\left[(1-\alpha)v(w) + \alpha d\right] \tag{6}$$

In the last expression, we wrote w_e as w to make the notation simpler.

38.4.3 The Reservation Wage

Suppose we can use (5) and (6) to solve for d and v.

(We will do this soon.)

We can then determine optimal behavior for the worker.

From (3), we see that an unemployed agent accepts current offer w if $v(w) \geq u(c) + \beta d$.

This means precisely that the value of accepting is higher than the expected value of rejecting.

It is clear that v is (at least weakly) increasing in w, since the agent is never made worse off by a higher wage offer.

Hence, we can express the optimal choice as accepting wage offer w if and only if

$$w \geq \bar{w} \quad \text{where} \quad \bar{w} \text{ solves } v(\bar{w}) = u(c) + \beta d$$

38.4.4 Solving the Bellman Equations

We'll use the same iterative approach to solving the Bellman equations that we adopted in the first job search lecture.

Here this amounts to

1. make guesses for d and v
2. plug these guesses into the right-hand sides of (5) and (6)
3. update the left-hand sides from this rule and then repeat

In other words, we are iterating using the rules

$$d_{n+1} = \sum_{w' \in \mathbb{W}} \max \{v_n(w'),\, u(c) + \beta d_n\}\, q(w') \tag{7}$$

$$v_{n+1}(w) = u(w) + \beta \left[(1-\alpha) v_n(w) + \alpha d_n\right] \tag{8}$$

starting from some initial conditions d_0, v_0.

As before, the system always converges to the true solutions—in this case, the v and d that solve (5) and (6).

(A proof can be obtained via the Banach contraction mapping theorem.)

38.5 Implementation

Let's implement this iterative process.

In the code, you'll see that we use a class to store the various parameters and other objects associated with a given model.

This helps to tidy up the code and provides an object that's easy to pass to functions.

The default utility function is a CRRA utility function

```
In [3]: @njit
        def u(c, σ=2.0):
            return (c**(1 - σ) - 1) / (1 - σ)
```

Also, here's a default wage distribution, based around the BetaBinomial distribution:

```
In [4]: n = 60                                  # n possible outcomes for w
        w_default = np.linspace(10, 20, n)      # wages between 10 and 20
        a, b = 600, 400                         # shape parameters
        dist = BetaBinomial(n-1, a, b)
        q_default = dist.pdf()
```

Here's our jitted class for the McCall model with separation.

```
In [5]: mccall_data = [
            ('α', float64),      # job separation rate
            ('β', float64),      # discount factor
            ('c', float64),      # unemployment compensation
            ('w', float64[:]),   # list of wage values
            ('q', float64[:])    # pmf of random variable w
        ]

        @jitclass(mccall_data)
        class McCallModel:
            """
            Stores the parameters and functions associated with a given model.
            """

            def __init__(self, α=0.2, β=0.98, c=6.0, w=w_default, q=q_default):

                self.α, self.β, self.c, self.w, self.q = α, β, c, w, q


            def update(self, v, d):

                α, β, c, w, q = self.α, self.β, self.c, self.w, self.q

                v_new = np.empty_like(v)

                for i in range(len(w)):
                    v_new[i] = u(w[i]) + β * ((1 - α) * v[i] + α * d)

                d_new = np.sum(np.maximum(v, u(c) + β * d) * q)

                return v_new, d_new
```

Now we iterate until successive realizations are closer together than some small tolerance level.

We then return the current iterate as an approximate solution.

```
In [6]: @njit
        def solve_model(mcm, tol=1e-5, max_iter=2000):
            """
            Iterates to convergence on the Bellman equations

            * mcm is an instance of McCallModel
            """

            v = np.ones_like(mcm.w)    # Initial guess of v
            d = 1                      # Initial guess of d
            i = 0
            error = tol + 1

            while error > tol and i < max_iter:
                v_new, d_new = mcm.update(v, d)
                error_1 = np.max(np.abs(v_new - v))
                error_2 = np.abs(d_new - d)
                error = max(error_1, error_2)
```

```
        v = v_new
        d = d_new
        i += 1

    return v, d
```

38.5.1 The Reservation Wage: First Pass

The optimal choice of the agent is summarized by the reservation wage.

As discussed above, the reservation wage is the $\bar{w}$ that solves $v(\bar{w}) = h$ where $h := u(c) + \beta d$ is the continuation value.

Let's compare v and h to see what they look like.

We'll use the default parameterizations found in the code above.

```
In [7]: mcm = McCallModel()
        v, d = solve_model(mcm)
        h = u(mcm.c) + mcm.β * d

        fig, ax = plt.subplots()

        ax.plot(mcm.w, v, 'b-', lw=2, alpha=0.7, label='$v$')
        ax.plot(mcm.w, [h] * len(mcm.w),
                'g-', lw=2, alpha=0.7, label='$h$')
        ax.set_xlim(min(mcm.w), max(mcm.w))
        ax.legend()

        plt.show()
```

The value v is increasing because higher w generates a higher wage flow conditional on staying employed.

38.5.2 The Reservation Wage: Computation

Here's a function `compute_reservation_wage` that takes an instance of `McCallModel` and returns the associated reservation wage.

In [8]:
```
@njit
def compute_reservation_wage(mcm):
    """
    Computes the reservation wage of an instance of the McCall model
    by finding the smallest w such that v(w) >= h.

    If no such w exists, then w_bar is set to np.inf.
    """

    v, d = solve_model(mcm)
    h = u(mcm.c) + mcm.β * d

    w_bar = np.inf
    for i, wage in enumerate(mcm.w):
        if v[i] > h:
            w_bar = wage
            break

    return w_bar
```

Next we will investigate how the reservation wage varies with parameters.

38.6 Impact of Parameters

In each instance below, we'll show you a figure and then ask you to reproduce it in the exercises.

38.6.1 The Reservation Wage and Unemployment Compensation

First, let's look at how $\bar{w}$ varies with unemployment compensation.

In the figure below, we use the default parameters in the `McCallModel` class, apart from c (which takes the values given on the horizontal axis)

As expected, higher unemployment compensation causes the worker to hold out for higher wages.

In effect, the cost of continuing job search is reduced.

38.6.2 The Reservation Wage and Discounting

Next, let's investigate how $\bar{w}$ varies with the discount factor.

The next figure plots the reservation wage associated with different values of β

Again, the results are intuitive: More patient workers will hold out for higher wages.

38.6.3 The Reservation Wage and Job Destruction

Finally, let's look at how $\bar{w}$ varies with the job separation rate α.

Higher α translates to a greater chance that a worker will face termination in each period once employed.

Once more, the results are in line with our intuition.

If the separation rate is high, then the benefit of holding out for a higher wage falls.

Hence the reservation wage is lower.

38.7 Exercises

38.7.1 Exercise 1

Reproduce all the reservation wage figures shown above.

Regarding the values on the horizontal axis, use

```
In [9]: grid_size = 25
        c_vals = np.linspace(2, 12, grid_size)          # unemployment compensation
        beta_vals = np.linspace(0.8, 0.99, grid_size)   # discount factors
        alpha_vals = np.linspace(0.05, 0.5, grid_size) # separation rate
```

38.8 Solutions

38.8.1 Exercise 1

Here's the first figure.

```
In [10]: mcm = McCallModel()

         w_bar_vals = np.empty_like(c_vals)

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

```
for i, c in enumerate(c_vals):
    mcm.c = c
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='unemployment compensation',
       ylabel='reservation wage')
ax.plot(c_vals, w_bar_vals, label=r'$\bar w$ as a function of $c$')
ax.legend()

plt.show()
```

Here's the second one.

```
In [11]: fig, ax = plt.subplots()

         for i, β in enumerate(beta_vals):
             mcm.β = β
             w_bar = compute_reservation_wage(mcm)
             w_bar_vals[i] = w_bar

         ax.set(xlabel='discount factor', ylabel='reservation wage')
         ax.plot(beta_vals, w_bar_vals, label=r'$\bar w$ as a function of $\beta$')
         ax.legend()

         plt.show()
```

Here's the third.

```
In [12]: fig, ax = plt.subplots()

         for i, α in enumerate(alpha_vals):
             mcm.α = α
             w_bar = compute_reservation_wage(mcm)
             w_bar_vals[i] = w_bar

         ax.set(xlabel='separation rate', ylabel='reservation wage')
         ax.plot(alpha_vals, w_bar_vals, label=r'$\bar w$ as a function of
↪$\alpha$')
         ax.legend()

         plt.show()
```

16.0
15.5
15.0
14.5
14.0
reservation wage
0.1
0.2
0.3
0.4
0.5
separation rate
$\bar{w}$ as a function of α

Chapter 39

Job Search III: Fitted Value Function Iteration

39.1 Contents

- Overview 39.2
- The Algorithm 39.3
- Implementation 39.4
- Exercises 39.5
- Solutions 39.6

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

```
In [1]: !pip install --upgrade quantecon
        !pip install --upgrade interpolation
```

39.2 Overview

In this lecture we again study the McCall job search model with separation, but now with a continuous wage distribution.

While we already considered continuous wage distributions briefly in the exercises of the first job search lecture, the change was relatively trivial in that case.

This is because we were able to reduce the problem to solving for a single scalar value (the continuation value).

Here, with separation, the change is less trivial, since a continuous wage distribution leads to an uncountably infinite state space.

The infinite state space leads to additional challenges, particularly when it comes to applying value function iteration (VFI).

These challenges will lead us to modify VFI by adding an interpolation step.

The combination of VFI and this interpolation step is called **fitted value function iteration** (fitted VFI).

Fitted VFI is very common in practice, so we will take some time to work through the details.

We will use the following imports:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

        import quantecon as qe
        from interpolation import interp
        from numpy.random import randn
        from numba import njit, jitclass, prange, float64, int32
```

39.3 The Algorithm

The model is the same as the McCall model with job separation we studied before, except that the wage offer distribution is continuous.

We are going to start with the two Bellman equations we obtained for the model with job separation after a simplifying transformation.

Modified to accommodate continuous wage draws, they take the following form:

$$d = \int \max\{v(w'),\, u(c) + \beta d\}\, q(w') dw' \tag{1}$$

and

$$v(w) = u(w) + \beta\left[(1-\alpha)v(w) + \alpha d\right] \tag{2}$$

The unknowns here are the function v and the scalar d.

The difference between these and the pair of Bellman equations we previously worked on are

1. in (1), what used to be a sum over a finite number of wage values is an integral over an infinite set.
2. The function v in (2) is defined over all $w \in \mathbb{R}_+$.

The function q in (1) is the density of the wage offer distribution.

Its support is taken as equal to $\mathbb{R}_+$.

39.3.1 Value Function Iteration

In theory, we should now proceed as follows:

1. Begin with a guess v, d for the solutions to (1)–(2).
2. Plug v, d into the right hand side of (1)–(2) and compute the left hand side to obtain updates v', d'
3. Unless some stopping condition is satisfied, set $(v, d) = (v', d')$ and go to step 2.

However, there is a problem we must confront before we implement this procedure: The iterates of the value function can neither be calculated exactly nor stored on a computer.

To see the issue, consider (2).

Even if v is a known function, the only way to store its update v' is to record its value $v'(w)$ for every $w \in \mathbb{R}_+$.

Clearly, this is impossible.

39.3.2 Fitted Value Function Iteration

What we will do instead is use **fitted value function iteration**.

The procedure is as follows:

Let a current guess v be given.

Now we record the value of the function v' at only finitely many "grid" points $w_1 < w_2 < \cdots < w_I$ and then reconstruct v' from this information when required.

More precisely, the algorithm will be

1. Begin with an array $\mathbf{v}$ representing the values of an initial guess of the value function on some grid points $\{w_i\}$.
2. Build a function v on the state space $\mathbb{R}_+$ by interpolation or approximation, based on $\mathbf{v}$ and $\{w_i\}$.
3. Obtain and record the samples of the updated function $v'(w_i)$ on each grid point w_i.
4. Unless some stopping condition is satisfied, take this as the new array and go to step 1.

How should we go about step 2?

This is a problem of function approximation, and there are many ways to approach it.

What's important here is that the function approximation scheme must not only produce a good approximation to each v, but also that it combines well with the broader iteration algorithm described above.

One good choice from both respects is continuous piecewise linear interpolation.

This method

1. combines well with value function iteration (see., e.g., [64] or [148]) and
2. preserves useful shape properties such as monotonicity and concavity/convexity.

Linear interpolation will be implemented using a JIT-aware Python interpolation library called interpolation.py.

The next figure illustrates piecewise linear interpolation of an arbitrary function on grid points $0, 0.2, 0.4, 0.6, 0.8, 1$.

```
In [3]: def f(x):
            y1 = 2 * np.cos(6 * x) + np.sin(14 * x)
```

```
    return y1 + 2.5

c_grid = np.linspace(0, 1, 6)
f_grid = np.linspace(0, 1, 150)

def Af(x):
    return interp(c_grid, f(c_grid), x)

fig, ax = plt.subplots()

ax.plot(f_grid, f(f_grid), 'b-', label='true function')
ax.plot(f_grid, Af(f_grid), 'g-', label='linear approximation')
ax.vlines(c_grid, c_grid * 0, f(c_grid), linestyle='dashed', alpha=0.5)

ax.legend(loc="upper center")

ax.set(xlim=(0, 1), ylim=(0, 6))
plt.show()
```

39.4 Implementation

The first step is to build a jitted class for the McCall model with separation and a continuous wage offer distribution.

We will take the utility function to be the log function for this application, with $u(c) = \ln c$.

We will adopt the lognormal distribution for wages, with $w = \exp(\mu + \sigma z)$ when z is standard normal and μ, σ are parameters.

```
In [4]: @njit
        def lognormal_draws(n=1000, μ=2.5, σ=0.5, seed=1234):
```

```
    np.random.seed(seed)
    z = np.random.randn(n)
    w_draws = np.exp(μ + σ * z)
    return w_draws
```

Here's our class.

```
In [5]: mccall_data_continuous = [
            ('c', float64),          # unemployment compensation
            ('α', float64),          # job separation rate
            ('β', float64),          # discount factor
            ('σ', float64),          # scale parameter in lognormal distribution
            ('μ', float64),          # location parameter in lognormal distribution
            ('w_grid', float64[:]),  # grid of points for fitted VFI
            ('w_draws', float64[:])  # draws of wages for Monte Carlo
        ]

        @jitclass(mccall_data_continuous)
        class McCallModelContinuous:

            def __init__(self,
                         c=1,
                         α=0.1,
                         β=0.96,
                         grid_min=1e-10,
                         grid_max=5,
                         grid_size=100,
                         w_draws=lognormal_draws()):

                self.c, self.α, self.β = c, α, β

                self.w_grid = np.linspace(grid_min, grid_max, grid_size)
                self.w_draws = w_draws

            def update(self, v, d):

                # Simplify names
                c, α, β, σ, μ = self.c, self.α, self.β, self.σ, self.μ
                w = self.w_grid
                u = lambda x: np.log(x)

                # Interpolate array represented value function
                vf = lambda x: interp(w, v, x)

                # Update d using Monte Carlo to evaluate integral
                d_new = np.mean(np.maximum(vf(self.w_draws), u(c) + β * d))

                # Update v
                v_new = u(w) + β * ((1 - α) * v + α * d)

                return v_new, d_new
```

We then return the current iterate as an approximate solution.

```
In [6]: @njit
        def solve_model(mcm, tol=1e-5, max_iter=2000):
```

```
    """
    Iterates to convergence on the Bellman equations

    * mcm is an instance of McCallModel
    """

    v = np.ones_like(mcm.w_grid)    # Initial guess of v
    d = 1                           # Initial guess of d
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        v_new, d_new = mcm.update(v, d)
        error_1 = np.max(np.abs(v_new - v))
        error_2 = np.abs(d_new - d)
        error = max(error_1, error_2)
        v = v_new
        d = d_new
        i += 1

    return v, d
```

Here's a function `compute_reservation_wage` that takes an instance of `McCallModelContinuous` and returns the associated reservation wage.

If $v(w) < h$ for all w, then the function returns np.inf

```
In [7]: @njit
        def compute_reservation_wage(mcm):
            """
            Computes the reservation wage of an instance of the McCall model
            by finding the smallest w such that v(w) >= h.

            If no such w exists, then w_bar is set to np.inf.
            """
            u = lambda x: np.log(x)

            v, d = solve_model(mcm)
            h = u(mcm.c) + mcm.β * d

            w_bar = np.inf
            for i, wage in enumerate(mcm.w_grid):
                if v[i] > h:
                    w_bar = wage
                    break

            return w_bar
```

The exercises ask you to explore the solution and how it changes with parameters.

39.5 Exercises

39.5.1 Exercise 1

Use the code above to explore what happens to the reservation wage when the wage parameter μ changes.

Use the default parameters and μ in `mu_vals = np.linspace(0.0, 2.0, 15)`

Is the impact on the reservation wage as you expected?

39.5.2 Exercise 2

Let us now consider how the agent responds to an increase in volatility.

To try to understand this, compute the reservation wage when the wage offer distribution is uniform on $(m - s, m + s)$ and s varies.

The idea here is that we are holding the mean constant and spreading the support.

(This is a form of *mean-preserving spread*.)

Use `s_vals = np.linspace(1.0, 2.0, 15)` and `m = 2.0`.

State how you expect the reservation wage vary with s.

Now compute it. Is this as you expected?

39.6 Solutions

39.6.1 Exercise 1

Here is one solution.

In [8]:
```
mcm = McCallModelContinuous()
mu_vals = np.linspace(0.0, 2.0, 15)
w_bar_vals = np.empty_like(mu_vals)

fig, ax = plt.subplots()

for i, m in enumerate(mu_vals):
    mcm.w_draws = lognormal_draws(μ=m)
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='mean', ylabel='reservation wage')
ax.plot(mu_vals, w_bar_vals, label=r'$\bar w$ as a function of $\mu$')
ax.legend()

plt.show()
```

Not surprisingly, the agent is more inclined to wait when the distribution of offers shifts to the right.

39.6.2 Exercise 2

Here is one solution.

```
In [9]: mcm = McCallModelContinuous()
        s_vals = np.linspace(1.0, 2.0, 15)
        m = 2.0
        w_bar_vals = np.empty_like(s_vals)

        fig, ax = plt.subplots()

        for i, s in enumerate(s_vals):
            a, b = m - s, m + s
            mcm.w_draws = np.random.uniform(low=a, high=b, size=10_000)
            w_bar = compute_reservation_wage(mcm)
            w_bar_vals[i] = w_bar

        ax.set(xlabel='volatility', ylabel='reservation wage')
        ax.plot(s_vals, w_bar_vals, label=r'$\bar w$ as a function of wage
↪volatility')
        ax.legend()

        plt.show()
```

The reservation wage increases with volatility.

One might think that higher volatility would make the agent more inclined to take a given offer, since doing so represents certainty and waiting represents risk.

But job search is like holding an option: the worker is only exposed to upside risk (since, in a free market, no one can force them to take a bad offer).

More volatility means higher upside potential, which encourages the agent to wait.

Chapter 40

Job Search IV: Correlated Wage Offers

40.1 Contents

- Overview 40.2
- The Model 40.3
- Implementation 40.4
- Unemployment Duration 40.5
- Exercises 40.6
- Solutions 40.7

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

```
In [1]: !pip install --upgrade quantecon
        !pip install --upgrade interpolation
```

40.2 Overview

In this lecture we solve a McCall style job search model with persistent and transitory components to wages.

In other words, we relax the unrealistic assumption that randomness in wages is independent over time.

At the same time, we will go back to assuming that jobs are permanent and no separation occurs.

This is to keep the model relatively simple as we study the impact of correlation.

We will use the following imports:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

        import quantecon as qe
        from interpolation import interp
        from numpy.random import randn
        from numba import njit, jitclass, prange, float64
```

40.3 The Model

Wages at each point in time are given by

$$w_t = \exp(z_t) + y_t$$

where

$$y_t \sim \exp(\mu + s\zeta_t) \quad \text{and} \quad z_{t+1} = d + \rho z_t + \sigma \epsilon_{t+1}$$

Here $\{\zeta_t\}$ and $\{\epsilon_t\}$ are both IID and standard normal.

Here $\{y_t\}$ is a transitory component and $\{z_t\}$ is persistent.

As before, the worker can either

1. accept an offer and work permanently at that wage, or
2. take unemployment compensation c and wait till next period.

The value function satisfies the Bellman equation

$$v^*(w, z) = \max \left\{ \frac{u(w)}{1-\beta}, u(c) + \beta \, \mathbb{E}_z v^*(w', z') \right\}$$

In this express, u is a utility function and $\mathbb{E}_z$ is expectation of next period variables given current z.

The variable z enters as a state in the Bellman equation because its current value helps predict future wages.

40.3.1 A Simplification

There is a way that we can reduce dimensionality in this problem, which greatly accelerates computation.

To start, let f^* be the continuation value function, defined by

$$f^*(z) := u(c) + \beta \, \mathbb{E}_z v^*(w', z')$$

The Bellman equation can now be written

$$v^*(w, z) = \max \left\{ \frac{u(w)}{1-\beta}, \, f^*(z) \right\}$$

Combining the last two expressions, we see that the continuation value function satisfies

$$f^*(z) = u(c) + \beta \, \mathbb{E}_z \max \left\{ \frac{u(w')}{1-\beta}, f^*(z') \right\}$$

We'll solve this functional equation for f^* by introducing the operator

$$Qf(z) = u(c) + \beta \, \mathbb{E}_z \max \left\{ \frac{u(w')}{1-\beta}, f(z') \right\}$$

By construction, f^* is a fixed point of Q, in the sense that $Qf^* = f^*$.

Under mild assumptions, it can be shown that Q is a contraction mapping over a suitable space of continuous functions on $\mathbb{R}$.

By Banach's contraction mapping theorem, this means that f^* is the unique fixed point and we can calculate it by iterating with Q from any reasonable initial condition.

Once we have f^*, we can solve the search problem by stopping when the reward for accepting exceeds the continuation value, or

$$\frac{u(w)}{1-\beta} \geq f^*(z)$$

For utility we take $u(c) = \ln(c)$.

The reservation wage is the wage where equality holds in the last expression.

That is,

$$\bar{w}(z) := \exp(f^*(z)(1-\beta)) \tag{1}$$

Our main aim is to solve for the reservation rule and study its properties and implications.

40.4 Implementation

Let f be our initial guess of f^*.

When we iterate, we use the fitted value function iteration algorithm.

In particular, f and all subsequent iterates are stored as a vector of values on a grid.

These points are interpolated into a function as required, using piecewise linear interpolation.

The integral in the definition of Qf is calculated by Monte Carlo.

The following list helps Numba by providing some type information about the data we will work with.

```
In [3]: job_search_data = [
            ('μ', float64),              # transient shock log mean
            ('s', float64),              # transient shock log variance
            ('d', float64),              # shift coefficient of persistent state
           ('ρ', float64),             # correlation coefficient of persistent state
            ('σ', float64),              # state volatility
            ('β', float64),              # discount factor
            ('c', float64),              # unemployment compensation
            ('z_grid', float64[:]),      # grid over the state space
            ('e_draws', float64[:,:])    # Monte Carlo draws for integration
        ]
```

Here's a class that stores the data and the right hand side of the Bellman equation.

Default parameter values are embedded in the class.

In [4]:
```
@jitclass(job_search_data)
class JobSearch:

    def __init__(self,
                 μ=0.0,        # transient shock log mean
                 s=1.0,        # transient shock log variance
                 d=0.0,        # shift coefficient of persistent state
                 ρ=0.9,       # correlation coefficient of persistent state
                 σ=0.1,        # state volatility
                 β=0.98,       # discount factor
                 c=5,          # unemployment compensation
                 mc_size=1000,
                 grid_size=100):

        self.μ, self.s, self.d,  = μ, s, d,
        self.ρ, self.σ, self.β, self.c = ρ, σ, β, c

        # Set up grid
        z_mean = d / (1 - ρ)
        z_sd = np.sqrt(σ / (1 - ρ**2))
        k = 3  # std devs from mean
        a, b = z_mean - k * z_sd, z_mean + k * z_sd
        self.z_grid = np.linspace(a, b, grid_size)

        # Draw and store shocks
        np.random.seed(1234)
        self.e_draws = randn(2, mc_size)

    def parameters(self):
        """
        Return all parameters as a tuple.
        """
        return self.μ, self.s, self.d, \
                self.ρ, self.σ, self.β, self.c
```

Next we implement the Q operator.

In [5]:
```
@njit(parallel=True)
def Q(js, f_in, f_out):
    """
    Apply the operator Q.

        * js is an instance of JobSearch
        * f_in and f_out are arrays that represent f and Qf respectively

    """

    μ, s, d, ρ, σ, β, c = js.parameters()
    M = js.e_draws.shape[1]

    for i in prange(len(js.z_grid)):
        z = js.z_grid[i]
        expectation = 0.0
        for m in range(M):
            e1, e2 = js.e_draws[:, m]
```

```
            z_next = d + ρ * z + σ * e1
            go_val = interp(js.z_grid, f_in, z_next)     # f(z')
            y_next = np.exp(μ + s * e2)                  # y' draw
            w_next = np.exp(z_next) + y_next             # w' draw
            stop_val = np.log(w_next) / (1 - β)
            expectation += max(stop_val, go_val)
        expectation = expectation / M
        f_out[i] = np.log(c) + β * expectation
```

Here's a function to compute an approximation to the fixed point of Q.

```
In [6]: def compute_fixed_point(js,
                                use_parallel=True,
                                tol=1e-4,
                                max_iter=1000,
                                verbose=True,
                                print_skip=25):

            f_init = np.log(js.c) * np.ones(len(js.z_grid))
            f_out = np.empty_like(f_init)

            # Set up loop
            f_in = f_init
            i = 0
            error = tol + 1

            while i < max_iter and error > tol:
                Q(js, f_in, f_out)
                error = np.max(np.abs(f_in - f_out))
                i += 1
                if verbose and i % print_skip == 0:
                    print(f"Error at iteration {i} is {error}.")
                f_in[:] = f_out

            if i == max_iter:
                print("Failed to converge!")

            if verbose and i < max_iter:
                print(f"\nConverged in {i} iterations.")

            return f_out
```

Let's try generating an instance and solving the model.

```
In [7]: js = JobSearch()

        qe.tic()
        f_star = compute_fixed_point(js, verbose=True)
        qe.toc()
```

```
    Error at iteration 25 is 0.6540143893175809.
Error at iteration 50 is 0.12643184012381425.
Error at iteration 75 is 0.030376323858035903.
Error at iteration 100 is 0.007581959253982973.
Error at iteration 125 is 0.0019085682645538782.
Error at iteration 150 is 0.00048173786846916755.
```

```
Error at iteration 175 is 0.000121400125664195.

Converged in 179 iterations.
TOC: Elapsed: 0:00:6.06
```

Out[7]: 6.063929080963135

Next we will compute and plot the reservation wage function defined in (1).

In [8]:
```
res_wage_function = np.exp(f_star * (1 - js.β))

fig, ax = plt.subplots()
ax.plot(js.z_grid, res_wage_function, label="reservation wage given $z$")
ax.set(xlabel="$z$", ylabel="wage")
ax.legend()
plt.show()
```

Notice that the reservation wage is increasing in the current state z.

This is because a higher state leads the agent to predict higher future wages, increasing the option value of waiting.

Let's try changing unemployment compensation and look at its impact on the reservation wage:

In [9]:
```
c_vals = 1, 2, 3

fig, ax = plt.subplots()

for c in c_vals:
```

```
    js = JobSearch(c=c)
    f_star = compute_fixed_point(js, verbose=False)
    res_wage_function = np.exp(f_star * (1 - js.β))
    ax.plot(js.z_grid, res_wage_function, label=f"$\\bar w$ at $c = {c}$")

ax.set(xlabel="$z$", ylabel="wage")
ax.legend()
plt.show()
```

As expected, higher unemployment compensation shifts the reservation wage up at all state values.

40.5 Unemployment Duration

Next we study how mean unemployment duration varies with unemployment compensation.

For simplicity we'll fix the initial state at $z_t = 0$.

```
In [10]: def compute_unemployment_duration(js, seed=1234):

             f_star = compute_fixed_point(js, verbose=False)
             μ, s, d, ρ, σ, β, c = js.parameters()
             z_grid = js.z_grid
             np.random.seed(seed)

             @njit
             def f_star_function(z):
                 return interp(z_grid, f_star, z)

             @njit
```

```
    def draw_tau(t_max=10_000):
        z = 0
        t = 0

        unemployed = True
        while unemployed and t < t_max:
            # draw current wage
            y = np.exp(μ + s * np.random.randn())
            w = np.exp(z) + y
            res_wage = np.exp(f_star_function(z) * (1 - β))
            # if optimal to stop, record t
            if w >= res_wage:
                unemployed = False
                τ = t
            # else increment data and state
            else:
                z = ρ * z + d + σ * np.random.randn()
                t += 1
        return τ

    @njit(parallel=True)
    def compute_expected_tau(num_reps=100_000):
        sum_value = 0
        for i in prange(num_reps):
            sum_value += draw_tau()
        return sum_value / num_reps

    return compute_expected_tau()
```

Let's test this out with some possible values for unemployment compensation.

```
In [11]: c_vals = np.linspace(1.0, 10.0, 8)
         durations = np.empty_like(c_vals)
         for i, c in enumerate(c_vals):
             js = JobSearch(c=c)
             τ = compute_unemployment_duration(js)
             durations[i] = τ
```

Here is a plot of the results.

```
In [12]: fig, ax = plt.subplots()
         ax.plot(c_vals, durations)
         ax.set_xlabel("unemployment compensation")
         ax.set_ylabel("mean unemployment duration")
         plt.show()
```

Not surprisingly, unemployment duration increases when unemployment compensation is higher.

This is because the value of waiting increases with unemployment compensation.

40.6 Exercises

40.6.1 Exercise 1

Investigate how mean unemployment duration varies with the discount factor β.

- What is your prior expectation?
- Do your results match up?

40.7 Solutions

40.7.1 Exercise 1

Here is one solution.

```
In [13]: beta_vals = np.linspace(0.94, 0.99, 8)
         durations = np.empty_like(beta_vals)
         for i, β in enumerate(beta_vals):
             js = JobSearch(β=β)
             τ = compute_unemployment_duration(js)
             durations[i] = τ
```

```
In [14]: fig, ax = plt.subplots()
         ax.plot(beta_vals, durations)
```

```
ax.set_xlabel("$\\beta$")
ax.set_ylabel("mean unemployment duration")
plt.show()
```

The figure shows that more patient individuals tend to wait longer before accepting an offer.

Chapter 41

Exchangeability and Bayesian Updating

41.1 Contents

- Overview 41.2
- Appendix 41.3

Co-author: Zejin Shi

41.2 Overview

This lecture studies an example of learning via Bayes' Law.

We touch on foundations of Bayesian statistical inference invented by Bruno DeFinetti [40].

The relevance of DeFinetti's work for economists is presented forcefully in chapter 11 of [100] by David Kreps.

The example that we study in this lecture is a key component of this lecture that augments the classic job search model of McCall [116] by presenting an unemployed worker with a statistical inference problem.

Here we create graphs that illustrate the role that a likelihood ratio plays in Bayes' Law.

We'll use such graphs to provide insights into the mechanics driving outcomes in this lecture about learning in an augmented McCall job search model.

Among other things, this lecture discusses connections between the statistical concepts of sequences of random variables that are

- independently and identically distributed
- exchangeable

Understanding the distinction between these concepts is essential for appreciating how Bayesian updating works in our example.

You can read about exchangeability here

Below, we'll often use

- W to denote a random variable

- w to denote a particular realization of a random variable W

Let's start with some imports:

```
In [1]: from numba import njit, vectorize
        from math import gamma
        import scipy.optimize as op
        from scipy.integrate import quad
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

41.2.1 Independently and Identically Distributed

We begin by looking at the notion of an **independently and identically distributed sequence** of random variables.

An independently and identically distributed sequence is often abbreviated as IID.

Two notions are involved, **independently** and **identically** distributed.

A sequence $W_0, W_1, \ldots$ is **independently distributed** if the joint probability density of the sequence is the **product** of the densities of the components of the sequence.

The sequence $W_0, W_1, \ldots$ is **independently and identically distributed** if in addition the marginal density of W_t is the same for all $t = 0, 1, \ldots$.

For example, let $p(W_0, W_1, \ldots)$ be the **joint density** of the sequence and let $p(W_t)$ be the **marginal density** for a particular W_t for all $t = 0, 1, \ldots$.

Then the joint density of the sequence $W_0, W_1, \ldots$ is IID if

$$p(W_0, W_1, \ldots) = p(W_0)p(W_1)\cdots$$

so that the joint density is the product of a sequence of identical marginal densities.

41.2.2 IID Means Past Observations Don't Tell Us Anything About Future Observations

If a sequence is random variables is IID, past information provides no information about future realizations.

In this sense, there is **nothing to learn** about the future from the past.

To understand these statements, let the joint distribution of a sequence of random variables $\{W_t\}_{t=0}^T$ that is not necessarily IID, be

$$p(W_T, W_{T-1}, \ldots, W_1, W_0)$$

Using the laws of probability, we can always factor such a joint density into a product of conditional densities:

$$\begin{aligned} p(W_T, W_{T-1}, \ldots, W_1, W_0) = & p(W_T|W_{t-1}, \ldots, W_0)p(W_{T-1}|W_{T-2}, \ldots, W_0)\cdots \\ & p(W_1|W_0)p(W_0) \end{aligned}$$

In general,

$$p(W_t|W_{t-1},\dots,W_0) \neq p(W_t)$$

which states that the **conditional density** on the left side does not equal the **marginal density** on the right side.

In the special IID case,

$$p(W_t|W_{t-1},\dots,W_0) = p(W_t)$$

and partial history $W_{t-1},\dots,W_0$ contains no information about the probability of W_t.

So in the IID case, there is **nothing to learn** about the densities of future random variables from past data.

In the general case, there is something go learn from past data.

We turn next to an instance of this general case in which there is something to learn from past data.

Please keep your eye out for **what** there is to learn from past data.

41.2.3 A Setting in Which Past Observations Are Informative

We now turn to a setting in which there **is** something to learn.

Let $\{W_t\}_{t=0}^{\infty}$ be a sequence of nonnegative scalar random variables with a joint probability distribution constructed as follows.

There are two distinct cumulative distribution functions F and G — with densities f and g for a nonnegative scalar random variable W.

Before the start of time, say at time $t = -1$, "nature" once and for all selects **either** f **or** g — and thereafter at each time $t \geq 0$ draws a random W from the selected distribution.

So the data are permanently generated as independently and identically distributed (IID) draws from **either** F **or** G.

We could say that *objectively* the probability that the data are generated as draws from F is either 0 or 1.

We now drop into this setting a decision maker who knows F and G and that nature picked one of them once and for all and then drew an IID sequence of draws from that distribution.

But our decision maker does not know which of the two distributions nature selected.

The decision maker summarizes his ignorance about this by picking a **subjective probability** $\tilde{\pi}$ and reasons as if nature had selected F with probability $\tilde{\pi} \in (0,1)$ and G with probability $1-\tilde{\pi}$.

Thus, we assume that the decision maker

- **knows** both F and G

- **doesnt't know** which of these two distributions that nature has drawn
- summarizing his ignorance by acting as if or **thinking** that nature chose distribution F with probability $\tilde{\pi} \in (0,1)$ and distribution G with probability $1-\tilde{\pi}$

- at date $t \geq 0$ has observed the partial history $w_t, w_{t-1}, \ldots, w_0$ of draws from the appropriate joint density of the partial history

But what do we mean by the *appropriate joint distribution*?

We'll discuss that next and in the process describe the concept of **exchangeability**.

41.2.4 Relationship Between IID and Exchangeable

Conditional on nature selecting F, the joint density of the sequence $W_0, W_1, \ldots$ is

$$f(W_0)f(W_1)\cdots$$

Conditional on nature selecting G, the joint density of the sequence $W_0, W_1, \ldots$ is

$$g(W_0)g(W_1)\cdots$$

Notice that **conditional on nature having selected** F, the sequence $W_0, W_1, \ldots$ is independently and identically distributed.

Furthermore, **conditional on nature having selected** G, the sequence $W_0, W_1, \ldots$ is also independently and identically distributed.

But what about the unconditional distribution?

The unconditional distribution of $W_0, W_1, \ldots$ is evidently

$$h(W_0, W_1, \ldots) \equiv \tilde{\pi}[f(W_0)f(W_1)\cdots] + (1-\tilde{\pi})[g(W_0)g(W_1)\cdots] \tag{1}$$

Under the unconditional distribution $h(W_0, W_1, \ldots)$, the sequence $W_0, W_1, \ldots$ is **not** independently and identically distributed.

To verify this claim, it is sufficient to notice, for example, that

$$h(w_0, w_1) = \tilde{\pi}f(w_0)f(w_1) + (1-\tilde{\pi})g(w_0)g(w_1) \neq (\tilde{\pi}f(w_0) + (1-\tilde{\pi})g(w_0))(\tilde{\pi}f(w_1) + (1-\tilde{\pi})g(w_1))$$

Thus, the conditional distribution

$$h(w_1|w_0) \equiv \frac{h(w_0, w_1)}{(\tilde{\pi}f(w_0) + (1-\tilde{\pi})g(w_0))} \neq (\tilde{\pi}f(w_1) + (1-\tilde{\pi})g(w_1))$$

This means that the realization w_0 contains information about w_1.

So there is something to learn.

But what and how?

41.2.5 Exchangeability

While the sequence $W_0, W_1, \ldots$ is not IID, it can be verified that it is **exchangeable**, which means that

$$h(w_0, w_1) = h(w_1, w_0)$$

and so on.

More generally, a sequence of random variables is said to be **exchangeable** if the joint probability distribution for the sequence does not change when the positions in the sequence in which finitely many of the random variables appear are altered.

Equation (1) represents our instance of an exchangeable joint density over a sequence of random variables as a **mixture** of two IID joint densities over a sequence of random variables.

For a Bayesian statistician, the mixing parameter $\tilde{\pi} \in (0, 1)$ has a special interpretation as a **prior probability** that nature selected probability distribution F.

DeFinetti [40] established a related representation of an exchangeable process created by mixing sequences of IID Bernoulli random variables with parameters θ and mixing probability $\pi(\theta)$ for a density $\pi(\theta)$ that a Bayesian statistician would interpret as a prior over the unknown Bernoulli paramter θ.

41.2.6 Bayes' Law

We noted above that in our example model there is something to learn about about the future from past data drawn from our particular instance of a process that is exchangeable but not IID.

But how can we learn?

And about what?

The answer to the *about what* question is about $\tilde{p}i$.

The answer to the *how* question is to use Bayes' Law.

Another way to say *use Bayes' Law* is to say *compute an appropriate conditional distribution.*

Let's dive into Bayes' Law in this context.

Let q represent the distribution that nature actually draws from w from and let

$$\pi = \mathbb{P}\{q = f\}$$

where we regard π as the decision maker's **subjective probability** (also called a **personal probability**.

Suppose that at $t \geq 0$, the decision maker has observed a history $w^t \equiv [w_t, w_{t-1}, \dots, w_0]$.

We let

$$\pi_t = \mathbb{P}\{q = f | w^t\}$$

where we adopt the convention

$$\pi_{-1} = \tilde{\pi}$$

The distribution of w_{t+1} conditional on w^t is then

$$\pi_t f + (1-\pi_t) g.$$

Bayes' rule for updating π_{t+1} is

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1-\pi_t) g(w_{t+1})} \tag{2}$$

The last expression follows from Bayes' rule, which tells us that

$$\mathbb{P}\{q = f \,|\, W = w\} = \frac{\mathbb{P}\{W = w \,|\, q = f\}\mathbb{P}\{q = f\}}{\mathbb{P}\{W = w\}} \quad \text{and} \quad \mathbb{P}\{W = w\} = \sum_{\omega \in \{f,g\}} \mathbb{P}\{W = w \,|\, q = \omega\}\mathbb{P}\{q = \omega\}$$

41.2.7 More Details about Bayesian Updating

Let's stare at and rearrange Bayes' Law as represented in equation (2) with the aim of understanding how the **posterior** π_{t+1} is influenced by the **prior** π_t and the **likelihood ratio**

$$l(w) = \frac{f(w)}{g(w)}$$

It is convenient for us to rewrite the updating rule (2) as

$$\pi_{t+1} = \frac{\pi_t f\left(w_{t+1}\right)}{\pi_t f\left(w_{t+1}\right) + \left(1-\pi_t\right) g\left(w_{t+1}\right)} = \frac{\pi_t \frac{f(w_{t+1})}{g(w_{t+1})}}{\pi_t \frac{f(w_{t+1})}{g(w_{t+1})} + \left(1-\pi_t\right)} = \frac{\pi_t l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + \left(1-\pi_t\right)}$$

This implies that

$$\frac{\pi_{t+1}}{\pi_t} = \frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + \left(1-\pi_t\right)} \begin{cases} > 1 & \text{if } l\left(w_{t+1}\right) > 1 \\ \leq 1 & \text{if } l\left(w_{t+1}\right) \leq 1 \end{cases} \tag{3}$$

Notice how the likelihood ratio and the prior interact to determine whether an observation w_{t+1} leads the decision maker to increase or decrease the subjective probability he/she attaches to distribution F.

When the likelihood ratio $l(w_{t+1})$ exceeds one, the observation w_{t+1} nudges the probability π put on distribution F upward, and when the likelihood ratio $l(w_{t+1})$ is less that one, the observation w_{t+1} nudges π downward.

Representation (3) is the foundation of the graphs that we'll use to display the dynamics of $\{\pi_t\}_{t=0}^{\infty}$ that are induced by Bayes' Law.

We'll plot $l\left(w\right)$ as a way to enlighten us about how learning – i.e., Bayesian updating of the probability π that nature has chosen distribution f – works.

To create the Python infrastructure to do our work for us, we construct a wrapper function that displays informative graphs given parameters of f and g.

```
In [2]: @vectorize
        def p(x, a, b):
```

```
    "The general beta distribution function."
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x ** (a-1) * (1 - x) ** (b-1)

def learning_example(F_a=1, F_b=1, G_a=3, G_b=1.2):
    """
    A wrapper function that displays the updating rule of belief π,
    given the parameters which specify F and G distributions.
    """

    f = njit(lambda x: p(x, F_a, F_b))
    g = njit(lambda x: p(x, G_a, G_b))

    # l(w) = f(w) / g(w)
    l = lambda w: f(w) / g(w)
    # objective function for solving l(w) = 1
    obj = lambda w: l(w) - 1

    x_grid = np.linspace(0, 1, 100)
    π_grid = np.linspace(1e-3, 1-1e-3, 100)

    w_max = 1
    w_grid = np.linspace(1e-12, w_max-1e-12, 100)

    # the mode of beta distribution
    # use this to divide w into two intervals for root finding
    G_mode = (G_a - 1) / (G_a + G_b - 2)
    roots = np.empty(2)
    roots[0] = op.root_scalar(obj, bracket=[1e-10, G_mode]).root
    roots[1] = op.root_scalar(obj, bracket=[G_mode, 1-1e-10]).root

    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 5))

    ax1.plot(l(w_grid), w_grid, label='$l$', lw=2)
    ax1.vlines(1., 0., 1., linestyle="--")
    ax1.hlines(roots, 0., 2., linestyle="--")
    ax1.set_xlim([0., 2.])
    ax1.legend(loc=4)
    ax1.set(xlabel='$l(w)=f(w)/g(w)$', ylabel='$w$')

    ax2.plot(f(x_grid), x_grid, label='$f$', lw=2)
    ax2.plot(g(x_grid), x_grid, label='$g$', lw=2)
    ax2.vlines(1., 0., 1., linestyle="--")
    ax2.hlines(roots, 0., 2., linestyle="--")
    ax2.legend(loc=4)
    ax2.set(xlabel='$f(w), g(w)$', ylabel='$w$')

    area1 = quad(f, 0, roots[0])[0]
    area2 = quad(g, roots[0], roots[1])[0]
    area3 = quad(f, roots[1], 1)[0]

    ax2.text((f(0) + f(roots[0])) / 4, roots[0] / 2, f"{area1: .3g}")
    ax2.fill_between([0, 1], 0, roots[0], color='blue', alpha=0.15)
    ax2.text(np.mean(g(roots)) / 2, np.mean(roots), f"{area2: .3g}")
    w_roots = np.linspace(roots[0], roots[1], 20)
    ax2.fill_betweenx(w_roots, 0, g(w_roots), color='orange', alpha=0.15)
    ax2.text((f(roots[1]) + f(1)) / 4, (roots[1] + 1) / 2, f"{area3: .3g}")
    ax2.fill_between([0, 1], roots[1], 1, color='blue', alpha=0.15)
```

```
W = np.arange(0.01, 0.99, 0.08)
Π = np.arange(0.01, 0.99, 0.08)

ΔW = np.zeros((len(W), len(Π)))
ΔΠ = np.empty((len(W), len(Π)))
for i, w in enumerate(W):
    for j, π in enumerate(Π):
        lw = l(w)
        ΔΠ[i, j] = π * (lw / (π * lw + 1 - π) - 1)

q = ax3.quiver(Π, W, ΔΠ, ΔW, scale=2, color='r', alpha=0.8)

ax3.fill_between(π_grid, 0, roots[0], color='blue', alpha=0.15)
ax3.fill_between(π_grid, roots[0], roots[1], color='green', alpha=0.15)
ax3.fill_between(π_grid, roots[1], w_max, color='blue', alpha=0.15)
ax3.hlines(roots, 0., 1., linestyle="--")
ax3.set(xlabel='$\pi$', ylabel='$w$')
ax3.grid()

plt.show()
```

Now we'll create a group of graphs designed to illustrate the dynamics induced by Bayes' Law.

We'll begin with the default values of various objects, then change them in a subsequent example.

```
In [3]: learning_example()
```

Please look at the three graphs above created for an instance in which f is a uniform distribution on $[0, 1]$ (i.e., a Beta distribution with parameters $F_a = 1, F_b = 1$, while g is a Beta distribution with the default parameter values $G_a = 3, G_b = 1.2$.

The graph in the left plots the likehood ratio $l(w)$ on the coordinate axis against w on the coordinate axis.

The middle graph plots both $f(w)$ and $g(w)$ against w, with the horizontal dotted lines showing values of w at which the likelihood ratio equals 1.

The graph on the right side plots arrows to the right that show when Bayes' Law makes π increase and arrows to the left that show when Bayes' Law make π decrease.

Notice how the length of the arrows, which show the magnitude of the force from Bayes' Law

impelling π to change, depend on both the prior probability π on the ordinate axis and the evidence in the form of the current draw of w on the coordinate axis.

The fractions in the colored areas of the middle graphs are probabilities under F and G, respectively, that realizations of w fall into the interval that updates the belief π in a correct direction (i.e., toward 0 when G is the true distribution, and towards 1 when F is the true distribution).

For example, in the above example, under true distribution F, π will be updated toward 0 if w falls into the interval $[0.524, 0.999]$, which occurs with probability $1 - .524 = .476$ under F. But this would occur with probability 0.816 if G were the true distribution. The fraction 0.816 in the orange region is the integral of $g(w)$ over this interval.

Next we use our code to create graphs for another instance of our model.

We keep F the same as in the preceding instance, namely a uniform distribution, but now assume that G is a Beta distribution with parameters $G_a = 2, G_b = 1.6$.

```
In [4]: learning_example(G_a=2, G_b=1.6)
```

Notice how the likelihood ratio, the middle graph, and the arrows compare with the previous instance of our example.

41.3 Appendix

41.3.1 Sample Paths of π_t

Now we'll have some fun by plotting multiple realizations of sample paths of π_t under two possible assumptions about nature's choice of distribution:

- that nature permanently draws from F
- that nature permanently draws from G

Outcomes depend on a peculiar property of likelihood ratio processes that are discussed in this lecture

To do this, we create some Python code.

```
In [5]: def function_factory(F_a=1, F_b=1, G_a=3, G_b=1.2):

            # define f and g
            f = njit(lambda x: p(x, F_a, F_b))
```

```
    g = njit(lambda x: p(x, G_a, G_b))

    @njit
    def update(a, b, π):
        "Update π by drawing from beta distribution with parameters a and b"

        # Draw
        w = np.random.beta(a, b)

        # Update belief
        π = 1 / (1 + ((1 - π) * g(w)) / (π * f(w)))

        return π

    @njit
    def simulate_path(a, b, T=50):
        "Simulates a path of beliefs π with length T"

        π = np.empty(T+1)

        # initial condition
        π[0] = 0.5

        for t in range(1, T+1):
            π[t] = update(a, b, π[t-1])

        return π

    def simulate(a=1, b=1, T=50, N=200, display=True):
        "Simulates N paths of beliefs π with length T"

        π_paths = np.empty((N, T+1))
        if display:
            fig = plt.figure()

        for i in range(N):
            π_paths[i] = simulate_path(a=a, b=b, T=T)
            if display:
             plt.plot(range(T+1), π_paths[i], color='b', lw=0.8, alpha=0.5)

        if display:
            plt.show()

        return π_paths

    return simulate
```

```
In [6]: simulate = function_factory()
```

We begin by generating N simulated $\{\pi_t\}$ paths with T periods when the sequence is truly IID draws from F. We set the initial prior $\pi_{-1} = .5$.

```
In [7]: T = 50
```

```
In [8]: # when nature selects F
        π_paths_F = simulate(a=1, b=1, T=T, N=1000)
```

In the above graph we observe that for most paths $\pi_t \to 1$. So Bayes' Law evidently eventually discovers the truth for most of our paths.

Next, we generate paths with T periods when the sequence is truly IID draws from G. Again, we set the initial prior $\pi_{-1} = .5$.

```
In [9]: # when nature selects G
        π_paths_G = simulate(a=3, b=1.2, T=T, N=1000)
```

In the above graph we observe that now most paths $\pi_t \to 0$.

41.3.2 Rates of convergence

We study rates of convergence of π_t to 1 when nature generates the data as IID draws from F and of π_t to 0 when nature generates the data as IID draws from G.

We do this by averaging across simulated paths of $\{\pi_t\}_{t=0}^T$.

Using N simulated π_t paths, we compute $1 - \sum_{i=1}^N \pi_{i,t}$ at each t when the data are generated as draws from F and compute $\sum_{i=1}^N \pi_{i,t}$ when the data are generated as draws from G.

```
In [10]: plt.plot(range(T+1), 1 - np.mean(π_paths_F, 0), label='F generates')
         plt.plot(range(T+1), np.mean(π_paths_G, 0), label='G generates')
         plt.legend()
         plt.title("convergence");
```

From the above graph, rates of convergence appear not to depend on whether F or G generates the data.

41.3.3 Another Graph of Population Dynamics of π_t

More insights about the dynamics of $\{\pi_t\}$ can be gleaned by computing the following conditional expectations of $\frac{\pi_{t+1}}{\pi_t}$ as functions of π_t via integration with respect to the pertinent probability distribution:

$$
\begin{aligned}
E\left[\frac{\pi_{t+1}}{\pi_t} \,\middle|\, q=\omega, \pi_t\right] &= E\left[\frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right)+\left(1-\pi_t\right)} \,\middle|\, q=\omega, \pi_t\right], \\
&= \int_0^1 \frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right)+\left(1-\pi_t\right)} \omega\left(w_{t+1}\right) d w_{t+1}
\end{aligned}
$$

where $\omega = f, g$.

The following code approximates the integral above:

```
In [11]: def expected_ratio(F_a=1, F_b=1, G_a=3, G_b=1.2):

             # define f and g
             f = njit(lambda x: p(x, F_a, F_b))
             g = njit(lambda x: p(x, G_a, G_b))

             l = lambda w: f(w) / g(w)
             integrand_f = lambda w, π: f(w) * l(w) / (π * l(w) + 1 - π)
             integrand_g = lambda w, π: g(w) * l(w) / (π * l(w) + 1 - π)

             π_grid = np.linspace(0.02, 0.98, 100)

             expected_rario = np.empty(len(π_grid))
             for q, inte in zip(["f", "g"], [integrand_f, integrand_g]):
                 for i, π in enumerate(π_grid):
                     expected_rario[i]= quad(inte, 0, 1, args=(π,))[0]
                 plt.plot(π_grid, expected_rario, label=f"{q} generates")

             plt.hlines(1, 0, 1, linestyle="--")
             plt.xlabel("$π_t$")
             plt.ylabel("$E[\pi_{t+1}/\pi_t]$")
             plt.legend()

             plt.show()
```

First, consider the case where $F_a = F_b = 1$ and $G_a = 3, G_b = 1.2$.

```
In [12]: expected_ratio()
```

The above graphs shows that when F generates the data, π_t on average always heads north, while when G generates the data, π_t heads south.

Next, we'll look at a degenerate case in whcih f and g are identical beta distributions, and $F_a = G_a = 3, F_b = G_b = 1.2$.

In a sense, here there is nothing to learn.

```
In [13]: expected_ratio(F_a=3, F_b=1.2)
```

The above graph says that π_t is inert and would remain at its initial value.

Finally, let's look at a case in which f and g are neither very different nor identical, in particular one in which $F_a = 2, F_b = 1$ and $G_a = 3, G_b = 1.2$.

```
In [14]: expected_ratio(F_a=2, F_b=1, G_a=3, G_b=1.2)
```

f generates
g generates
1.150
1.125
1.100
1.075
1.050
1.025
1.000
0.975
$E[\pi_{t+1}/\pi_t]$
0.0
0.2
0.4
0.6
0.8
1.0
π_t

Chapter 42

Job Search V: Search with Learning

42.1 Contents

- Overview 42.2
- Model 42.3
- Take 1: Solution by VFI 42.4
- Take 2: A More Efficient Method 42.5
- Another Functional Equation 42.6
- Solving the RWFE 42.7
- Implementation 42.8
- Exercises 42.9
- Solutions 42.10
- Appendix A 42.11
- Appendix B 42.12
- Examples 42.13

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

In [1]:
```
!pip install --upgrade quantecon
!pip install interpolation
```

42.2 Overview

In this lecture, we consider an extension of the previously studied job search model of McCall [116].

We'll build on a model of Bayesian learning discussed in this lecture on the topic of exchangeability and its relationship to the concept of IID (identically and independently distributed) random variables and to Bayesian updating.

In the McCall model, an unemployed worker decides when to accept a permanent job at a specific fixed wage, given

- his or her discount factor
- the level of unemployment compensation
- the distribution from which wage offers are drawn

In the version considered below, the wage distribution is unknown and must be learned.

- The following is based on the presentation in [108], section 6.6.

Let's start with some imports

In [2]:
```
from numba import njit, prange, vectorize
from interpolation import mlinterp, interp
from math import gamma
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib import cm
import scipy.optimize as op
from scipy.stats import cumfreq, beta
```

42.2.1 Model Features

- Infinite horizon dynamic programming with two states and one binary control.
- Bayesian updating to learn the unknown distribution.

42.3 Model

Let's first review the basic McCall model [116] and then add the variation we want to consider.

42.3.1 The Basic McCall Model

Recall that, in the baseline model , an unemployed worker is presented in each period with a permanent job offer at wage W_t.

At time t, our worker either

1. accepts the offer and works permanently at constant wage W_t
2. rejects the offer, receives unemployment compensation c and reconsiders next period

The wage sequence W_t is IID and generated from known density q.

The worker aims to maximize the expected discounted sum of earnings $\mathbb{E}\sum_{t=0}^{\infty}\beta^t y_t$ The function V satisfies the recursion

$$v(w) = \max\left\{\frac{w}{1-\beta}, c + \beta\int v(w')q(w')dw'\right\} \tag{1}$$

The optimal policy has the form $\mathbf{1}\{w \geq \bar{w}\}$, where $\bar{w}$ is a constant called the *reservation wage*.

42.3.2 Offer Distribution Unknown

Now let's extend the model by considering the variation presented in [108], section 6.6.

The model is as above, apart from the fact that

- the density q is unknown
- the worker learns about q by starting with a prior and updating based on wage offers that he/she observes

The worker knows there are two possible distributions F and G — with densities f and g.

At the start of time, "nature" selects q to be either f or g — the wage distribution from which the entire sequence W_t will be drawn.

This choice is not observed by the worker, who puts prior probability π_0 on f being chosen.

Update rule: worker's time t estimate of the distribution is $\pi_t f + (1 - \pi_t)g$, where π_t updates via

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t) g(w_{t+1})} \tag{2}$$

This last expression follows from Bayes' rule, which tells us that

$$\mathbb{P}\{q = f \,|\, W = w\} = \frac{\mathbb{P}\{W = w \,|\, q = f\}\mathbb{P}\{q = f\}}{\mathbb{P}\{W = w\}} \quad \text{and} \quad \mathbb{P}\{W = w\} = \sum_{\omega \in \{f, g\}} \mathbb{P}\{W = w \,|\, q = \omega\}\mathbb{P}\{q = \omega\}$$

The fact that (2) is recursive allows us to progress to a recursive solution method.

Letting

$$q_\pi(w) := \pi f(w) + (1 - \pi) g(w) \quad \text{and} \quad \kappa(w, \pi) := \frac{\pi f(w)}{\pi f(w) + (1 - \pi) g(w)}$$

we can express the value function for the unemployed worker recursively as follows

$$v(w, \pi) = \max \left\{ \frac{w}{1 - \beta}, \, c + \beta \int v(w', \pi') \, q_\pi(w') \, dw' \right\} \quad \text{where} \quad \pi' = \kappa(w', \pi) \tag{3}$$

Notice that the current guess π is a state variable, since it affects the worker's perception of probabilities for future rewards.

42.3.3 Parameterization

Following section 6.6 of [108], our baseline parameterization will be

- f is $\text{Beta}(1, 1)$
- g is $\text{Beta}(3, 1.2)$
- $\beta = 0.95$ and $c = 0.3$

The densities f and g have the following shape

```
In [3]: @vectorize
        def p(x, a, b):
            r = gamma(a + b) / (gamma(a) * gamma(b))
            return r * x**(a-1) * (1 - x)**(b-1)
```

```
x_grid = np.linspace(0, 1, 100)
f = lambda x: p(x, 1, 1)
g = lambda x: p(x, 3, 1.2)

fig, ax = plt.subplots(figsize=(10, 8))
ax.plot(x_grid, f(x_grid), label='$f$', lw=2)
ax.plot(x_grid, g(x_grid), label='$g$', lw=2)

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

42.3.4 Looking Forward

What kind of optimal policy might result from (3) and the parameterization specified above?

Intuitively, if we accept at w_a and $w_a \leq w_b$, then — all other things being given — we should also accept at w_b.

This suggests a policy of accepting whenever w exceeds some threshold value $\bar{w}$.

But $\bar{w}$ should depend on π — in fact, it should be decreasing in π because

- f is a less attractive offer distribution than g
- larger π means more weight on f and less on g

Thus larger π depresses the worker's assessment of her future prospects, and relatively low current offers become more attractive.

Summary: We conjecture that the optimal policy is of the form $\mathbb{1}w \geq \bar{w}(\pi)$ for some decreasing function $\bar{w}$.

42.4 Take 1: Solution by VFI

Let's set about solving the model and see how our results match with our intuition.

We begin by solving via value function iteration (VFI), which is natural but ultimately turns out to be second best.

The class `SearchProblem` is used to store parameters and methods needed to compute optimal actions.

```
In [4]: class SearchProblem:
            """
            A class to store a given parameterization of the "offer distribution
            unknown" model.

            """

            def __init__(self,
                         β=0.95,            # Discount factor
                         c=0.3,             # Unemployment compensation
                         F_a=1,
                         F_b=1,
                         G_a=3,
                         G_b=1.2,
                         w_max=1,           # Maximum wage possible
                         w_grid_size=100,
                         π_grid_size=100,
                         mc_size=500):

                self.β, self.c, self.w_max = β, c, w_max

                self.f = njit(lambda x: p(x, F_a, F_b))
                self.g = njit(lambda x: p(x, G_a, G_b))

                self.π_min, self.π_max = 1e-3, 1-1e-3    # Avoids instability
                self.w_grid = np.linspace(0, w_max, w_grid_size)
                self.π_grid = np.linspace(self.π_min, self.π_max, π_grid_size)

                self.mc_size = mc_size

                self.w_f = np.random.beta(F_a, F_b, mc_size)
                self.w_g = np.random.beta(G_a, G_b, mc_size)
```

The following function takes an instance of this class and returns jitted versions of the Bellman operator `T`, and a `get_greedy()` function to compute the approximate optimal policy from a guess `v` of the value function

```
In [5]: def operator_factory(sp, parallel_flag=True):

            f, g = sp.f, sp.g
            w_f, w_g = sp.w_f, sp.w_g
            β, c = sp.β, sp.c
```

```
    mc_size = sp.mc_size
    w_grid, π_grid = sp.w_grid, sp.π_grid

    @njit
    def ϰ(w, π):
        """
        Updates π using Bayes' rule and the current wage observation w.
        """
        pf, pg = π * f(w), (1 - π) * g(w)
        π_new = pf / (pf + pg)

        return π_new

    @njit(parallel=parallel_flag)
    def T(v):
        """
        The Bellman operator.

        """
        v_func = lambda x, y: mlinterp((w_grid, π_grid), v, (x, y))
        v_new = np.empty_like(v)

        for i in prange(len(w_grid)):
            for j in prange(len(π_grid)):
                w = w_grid[i]
                π = π_grid[j]

                v_1 = w / (1 - β)

                integral_f, integral_g = 0.0, 0.0
                for m in prange(mc_size):
                    integral_f += v_func(w_f[m], ϰ(w_f[m], π))
                    integral_g += v_func(w_g[m], ϰ(w_g[m], π))
               integral = (π * integral_f + (1 - π) * integral_g) / mc_size

                v_2 = c + β * integral
                v_new[i, j] = max(v_1, v_2)

        return v_new

    @njit(parallel=parallel_flag)
    def get_greedy(v):
        """"
        Compute optimal actions taking v as the value function.

        """

        v_func = lambda x, y: mlinterp((w_grid, π_grid), v, (x, y))
        σ = np.empty_like(v)

        for i in prange(len(w_grid)):
            for j in prange(len(π_grid)):
                w = w_grid[i]
                π = π_grid[j]

                v_1 = w / (1 - β)

                integral_f, integral_g = 0.0, 0.0
```

```
                for m in prange(mc_size):
                    integral_f += v_func(w_f[m], ϰ(w_f[m], π))
                    integral_g += v_func(w_g[m], ϰ(w_g[m], π))
                integral = (π * integral_f + (1 - π) * integral_g) / mc_size

                v_2 = c + β * integral

                σ[i, j] = v_1 > v_2  # Evaluates to 1 or 0

            return σ

        return T, get_greedy
```

We will omit a detailed discussion of the code because there is a more efficient solution method that we will use later.

To solve the model we will use the following function that iterates using T to find a fixed point

```
In [6]: def solve_model(sp,
                        use_parallel=True,
                        tol=1e-4,
                        max_iter=1000,
                        verbose=True,
                        print_skip=5):

            """
            Solves for the value function

            * sp is an instance of SearchProblem
            """

            T, _ = operator_factory(sp, use_parallel)

            # Set up loop
            i = 0
            error = tol + 1
            m, n = len(sp.w_grid), len(sp.π_grid)

            # Initialize v
            v = np.zeros((m, n)) + sp.c / (1 - sp.β)

            while i < max_iter and error > tol:
                v_new = T(v)
                error = np.max(np.abs(v - v_new))
                i += 1
                if verbose and i % print_skip == 0:
                    print(f"Error at iteration {i} is {error}.")
                v = v_new

            if i == max_iter:
                print("Failed to converge!")

            if verbose and i < max_iter:
                print(f"\nConverged in {i} iterations.")


            return v_new
```

Let's look at solutions computed from value function iteration

In [7]:
```
sp = SearchProblem()
v_star = solve_model(sp)
fig, ax = plt.subplots(figsize=(6, 6))
ax.contourf(sp.π_grid, sp.w_grid, v_star, 12, alpha=0.6, cmap=cm.jet)
cs = ax.contour(sp.π_grid, sp.w_grid, v_star, 12, colors="black")
ax.clabel(cs, inline=1, fontsize=10)
ax.set(xlabel='$\pi$', ylabel='$w$')

plt.show()
```

```
    Error at iteration 5 is 0.6213817929052698.
Error at iteration 10 is 0.10385829667012914.
Error at iteration 15 is 0.022294799068692583.
Error at iteration 20 is 0.004930026818463773.
Error at iteration 25 is 0.0010903642993280727.
Error at iteration 30 is 0.0002411570040443678.

Converged in 33 iterations.
```

We will also plot the optimal policy

In [8]:
```
T, get_greedy = operator_factory(sp)
σ_star = get_greedy(v_star)

fig, ax = plt.subplots(figsize=(6, 6))
ax.contourf(sp.π_grid, sp.w_grid, σ_star, 1, alpha=0.6, cmap=cm.jet)
ax.contour(sp.π_grid, sp.w_grid, σ_star, 1, colors="black")
ax.set(xlabel='$\pi$', ylabel='$w$')

ax.text(0.5, 0.6, 'reject')
ax.text(0.7, 0.9, 'accept')

plt.show()
```

The results fit well with our intuition from section [#looking-forward}{looking forward}.

- The black line in the figure above corresponds to the function $\bar{w}(\pi)$ introduced there.
- It is decreasing as expected.

42.5 Take 2: A More Efficient Method

Let's consider another method to solve for the optimal policy.

We will use iteration with an operator that has the same contraction rate as the Bellman operator, but

- one dimensional rather than two dimensional
- no maximization step

As a consequence, the algorithm is orders of magnitude faster than VFI.

This section illustrates the point that when it comes to programming, a bit of mathematical analysis goes a long way.

42.6 Another Functional Equation

To begin, note that when $w = \bar{w}(\pi)$, the worker is indifferent between accepting and rejecting.

Hence the two choices on the right-hand side of (3) have equal value:

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int v(w', \pi')\, q_\pi(w')\, dw' \tag{4}$$

Together, (3) and (4) give

$$v(w, \pi) = \max\left\{\frac{w}{1-\beta}, \frac{\bar{w}(\pi)}{1-\beta}\right\} \tag{5}$$

Combining (4) and (5), we obtain

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int \max\left\{\frac{w'}{1-\beta}, \frac{\bar{w}(\pi')}{1-\beta}\right\} q_\pi(w')\, dw'$$

Multiplying by $1-\beta$, substituting in $\pi' = \kappa(w', \pi)$ and using $\circ$ for composition of functions yields

$$\bar{w}(\pi) = (1-\beta)c + \beta \int \max\left\{w', \bar{w} \circ \kappa(w', \pi)\right\}\, q_\pi(w')\, dw' \tag{6}$$

Equation (6) can be understood as a functional equation, where $\bar{w}$ is the unknown function.

- Let's call it the *reservation wage functional equation* (RWFE).
- The solution $\bar{w}$ to the RWFE is the object that we wish to compute.

42.7 Solving the RWFE

To solve the RWFE, we will first show that its solution is the fixed point of a contraction mapping.

To this end, let

- $b[0,1]$ be the bounded real-valued functions on $[0,1]$
- $\|\omega\| := \sup_{x \in [0,1]} |\omega(x)|$

Consider the operator Q mapping $\omega \in b[0,1]$ into $Q\omega \in b[0,1]$ via

$$(Q\omega)(\pi) = (1-\beta)c + \beta \int \max\{w', \omega \circ \kappa(w', \pi)\} \; q_\pi(w') \, dw' \tag{7}$$

Comparing (6) and (7), we see that the set of fixed points of Q exactly coincides with the set of solutions to the RWFE.

- If $Q\bar{w} = \bar{w}$ then $\bar{w}$ solves (6) and vice versa.

Moreover, for any $\omega, \omega' \in b[0,1]$, basic algebra and the triangle inequality for integrals tells us that

$$|(Q\omega)(\pi) - (Q\omega')(\pi)| \leq \beta \int |\max\{w', \omega \circ \kappa(w', \pi)\} - \max\{w', \omega' \circ \kappa(w', \pi)\}| \; q_\pi(w') \, dw' \tag{8}$$

Working case by case, it is easy to check that for real numbers a, b, c we always have

$$|\max\{a, b\} - \max\{a, c\}| \leq |b - c| \tag{9}$$

Combining (8) and (9) yields

$$|(Q\omega)(\pi) - (Q\omega')(\pi)| \leq \beta \int |\omega \circ \kappa(w', \pi) - \omega' \circ \kappa(w', \pi)| \; q_\pi(w') \, dw' \leq \beta \|\omega - \omega'\| \tag{10}$$

Taking the supremum over π now gives us

$$\|Q\omega - Q\omega'\| \leq \beta \|\omega - \omega'\| \tag{11}$$

In other words, Q is a contraction of modulus β on the complete metric space $(b[0,1], \|\cdot\|)$.

Hence

- A unique solution $\bar{w}$ to the RWFE exists in $b[0,1]$.
- $Q^k \omega \to \bar{w}$ uniformly as $k \to \infty$, for any $\omega \in b[0,1]$.

42.8 Implementation

The following function takes an instance of `SearchProblem` and returns the operator `Q`

```
In [9]: def Q_factory(sp, parallel_flag=True):

            f, g = sp.f, sp.g
            w_f, w_g = sp.w_f, sp.w_g
            β, c = sp.β, sp.c
            mc_size = sp.mc_size
            w_grid, π_grid = sp.w_grid, sp.π_grid

            @njit
            def ϰ(w, π):
                """
                Updates π using Bayes' rule and the current wage observation w.
                """
```

```
        pf, pg = π * f(w), (1 - π) * g(w)
        π_new = pf / (pf + pg)

        return π_new

    @njit(parallel=parallel_flag)
    def Q(ω):
        """

        Updates the reservation wage function guess ω via the operator
        Q.

        """
        ω_func = lambda p: interp(π_grid, ω, p)
        ω_new = np.empty_like(ω)

        for i in prange(len(π_grid)):
            π = π_grid[i]
            integral_f, integral_g = 0.0, 0.0

            for m in prange(mc_size):
                integral_f += max(w_f[m], ω_func(ϰ(w_f[m], π)))
                integral_g += max(w_g[m], ω_func(ϰ(w_g[m], π)))
            integral = (π * integral_f + (1 - π) * integral_g) / mc_size

            ω_new[i] = (1 - β) * c + β * integral

        return ω_new

    return Q
```

In the next exercise, you are asked to compute an approximation to $\bar{w}$.

42.9 Exercises

42.9.1 Exercise 1

Use the default parameters and `Q_factory` to compute an optimal policy.

Your result should coincide closely with the figure for the optimal policy [#odu-pol-vfi}{shown above}.

Try experimenting with different parameters, and confirm that the change in the optimal policy coincides with your intuition.

42.10 Solutions

42.10.1 Exercise 1

This code solves the "Offer Distribution Unknown" model by iterating on a guess of the reservation wage function.

You should find that the run time is shorter than that of the value function approach.

Similar to above, we set up a function to iterate with `Q` to find the fixed point

```
In [10]: def solve_wbar(sp,
                        use_parallel=True,
                        tol=1e-4,
                        max_iter=1000,
                        verbose=True,
                        print_skip=5):

             Q = Q_factory(sp, use_parallel)

             # Set up loop
             i = 0
             error = tol + 1
             m, n = len(sp.w_grid), len(sp.π_grid)

             # Initialize w
             w = np.ones_like(sp.π_grid)

             while i < max_iter and error > tol:
                 w_new = Q(w)
                 error = np.max(np.abs(w - w_new))
                 i += 1
                 if verbose and i % print_skip == 0:
                     print(f"Error at iteration {i} is {error}.")
                 w = w_new

             if i == max_iter:
                 print("Failed to converge!")

             if verbose and i < max_iter:
                 print(f"\nConverged in {i} iterations.")

             return w_new
```

The solution can be plotted as follows

```
In [11]: sp = SearchProblem()
         w_bar = solve_wbar(sp)

         fig, ax = plt.subplots(figsize=(9, 7))

         ax.plot(sp.π_grid, w_bar, color='k')
         ax.fill_between(sp.π_grid, 0, w_bar, color='blue', alpha=0.15)
         ax.fill_between(sp.π_grid, w_bar, sp.w_max, color='green', alpha=0.15)
         ax.text(0.5, 0.6, 'reject')
         ax.text(0.7, 0.9, 'accept')
         ax.set(xlabel='$\pi$', ylabel='$w$')
         ax.grid()
         plt.show()
```

```
    Error at iteration 5 is 0.021028277552939012.
Error at iteration 10 is 0.006120954180974025.
Error at iteration 15 is 0.0014954323647495826.
Error at iteration 20 is 0.0003420899319805937.
Error at iteration 25 is 7.627610193061685e-05.
```

```
Converged in 25 iterations.
```

42.11 Appendix A

The next piece of code generates a fun simulation to see what the effect of a change in the underlying distribution on the unemployment rate is.

At a point in the simulation, the distribution becomes significantly worse.

It takes a while for agents to learn this, and in the meantime, they are too optimistic and turn down too many jobs.

As a result, the unemployment rate spikes

```
In [12]: F_a, F_b, G_a, G_b = 1, 1, 3, 1.2

         sp = SearchProblem(F_a=F_a, F_b=F_b, G_a=G_a, G_b=G_b)
         f, g = sp.f, sp.g

         # Solve for reservation wage
         w_bar = solve_wbar(sp, verbose=False)

         # Interpolate reservation wage function
         π_grid = sp.π_grid
         w_func = njit(lambda x: interp(π_grid, w_bar, x))
```

```
@njit
def update(a, b, e, π):
    "Update e and π by drawing wage offer from beta distribution with
↪parameters a and
b"

    if e == False:
        w = np.random.beta(a, b)       # Draw random wage
        if w >= w_func(π):
            e = True                   # Take new job
        else:
            π = 1 / (1 + ((1 - π) * g(w)) / (π * f(w)))

    return e, π

@njit
def simulate_path(F_a=F_a,
                  F_b=F_b,
                  G_a=G_a,
                  G_b=G_b,
                  N=5000,       # Number of agents
                  T=600,        # Simulation length
                  d=200,        # Change date
                  s=0.025):     # Separation rate

   """Simulates path of employment for N number of works over T periods"""

    e = np.ones((N, T+1))
    π = np.ones((N, T+1)) * 1e-3

    a, b = G_a, G_b   # Initial distribution parameters

    for t in range(T+1):

        if t == d:
            a, b = F_a, F_b  # Change distribution parameters

        # Update each agent
        for n in range(N):
            if e[n, t] == 1:                    # If agent is currently
↪employment
                p = np.random.uniform(0, 1)
                if p <= s:                      # Randomly separate with
↪probability s
                    e[n, t] = 0

            new_e, new_π = update(a, b, e[n, t], π[n, t])
            e[n, t+1] = new_e
            π[n, t+1] = new_π

    return e[:, 1:]

d = 200  # Change distribution at time d
unemployment_rate = 1 - simulate_path(d=d).mean(axis=0)

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(unemployment_rate)
```

```
ax.axvline(d, color='r', alpha=0.6, label='Change date')
ax.set_xlabel('Time')
ax.set_title('Unemployment rate')
ax.legend()
plt.show()
```

42.12 Appendix B

In this appendix we provide more details about how Bayes' Law contributes to the workings of the model.

We present some graphs that bring out additional insights about how learning works.

We build on graphs proposed in this lecture.

In particular, we'll add actions of our searching worker to a key graph presented in that lecture.

To begin, we first define two functions for computing the empirical distributions of unemployment duration and π at the time of employment.

```
In [13]: @njit
         def empirical_dist(F_a, F_b, G_a, G_b, w_bar, π_grid,
                            N=10000, T=600):
             """
             Simulates population for computing empirical cumulative
             distribution of unemployment duration and π at time when
             the worker accepts the wage offer. For each job searching
             problem, we simulate for two cases that either f or g is
             the true offer distribution.
```

```
    Parameters
    ----------

    F_a, F_b, G_a, G_b : parameters of beta distributions F and G.
    w_bar : the reservation wage
    π_grid : grid points of π, for interpolation
    N : number of workers for simulation, optional
    T : maximum of time periods for simulation, optional

    Returns
    -------
    accpet_t : 2 by N ndarray. the empirical distribution of
               unemployment duration when f or g generates offers.
    accept_π : 2 by N ndarray. the empirical distribution of
               π at the time of employment when f or g generates offers.
    """

    accept_t = np.empty((2, N))
    accept_π = np.empty((2, N))

    # f or g generates offers
    for i, (a, b) in enumerate([(F_a, F_b), (G_a, G_b)]):
        # update each agent
        for n in range(N):

            # initial priori
            π = 0.5

            for t in range(T+1):

                # Draw random wage
                w = np.random.beta(a, b)
                lw = p(w, F_a, F_b) / p(w, G_a, G_b)
                π = π * lw / (π * lw + 1 - π)

                # move to next agent if accepts
                if w >= interp(π_grid, w_bar, π):
                    break

            # record the unemployment duration
            # and π at the time of acceptance
            accept_t[i, n] = t
            accept_π[i, n] = π

    return accept_t, accept_π

def cumfreq_x(res):
    """
    A helper function for calculating the x grids of
    the cumulative frequency histogram.
    """

    cumcount = res.cumcount
    lowerlimit, binsize = res.lowerlimit, res.binsize

    x = lowerlimit + np.linspace(0, binsize*cumcount.size, cumcount.size)
```

```
    return x
```

Now we define a wrapper function for analyzing job search models with learning under different parameterizations.

The wrapper takes parameters of beta distributions and unemployment compensation as inputs and then displays various things we want to know to interpret the solution of our search model.

In addition, it computes empirical cumulative distributions of two key objects.

```
In [14]: def job_search_example(F_a=1, F_b=1, G_a=3, G_b=1.2, c=0.3):
             """
             Given the parameters that specify F and G distributions,
             calculate and display the rejection and acceptance area,
             the evolution of belief π, and the probability of accepting
             an offer at different π level, and simulate and calculate
             the empirical cumulative distribution of the duration of
             unemployment and π at the time the worker accepts the offer.
             """

             # construct a search problem
             sp = SearchProblem(F_a=F_a, F_b=F_b, G_a=G_a, G_b=G_b, c=c)
             f, g = sp.f, sp.g
             π_grid = sp.π_grid

             # Solve for reservation wage
             w_bar = solve_wbar(sp, verbose=False)

             # l(w) = f(w) / g(w)
             l = lambda w: f(w) / g(w)
             # objective function for solving l(w) = 1
             obj = lambda w: l(w) - 1.

             # the mode of beta distribution
             # use this to divide w into two intervals for root finding
             G_mode = (G_a - 1) / (G_a + G_b - 2)
             roots = np.empty(2)
             roots[0] = op.root_scalar(obj, bracket=[1e-10, G_mode]).root
             roots[1] = op.root_scalar(obj, bracket=[G_mode, 1-1e-10]).root

             fig, axs = plt.subplots(2, 2, figsize=(12, 9))

             # part 1: display the details of the model settings and some results
             w_grid = np.linspace(1e-12, 1-1e-12, 100)

             axs[0, 0].plot(l(w_grid), w_grid, label='$l$', lw=2)
             axs[0, 0].vlines(1., 0., 1., linestyle="--")
             axs[0, 0].hlines(roots, 0., 2., linestyle="--")
             axs[0, 0].set_xlim([0., 2.])
             axs[0, 0].legend(loc=4)
             axs[0, 0].set(xlabel='$l(w)=f(w)/g(w)$', ylabel='$w$')

             axs[0, 1].plot(sp.π_grid, w_bar, color='k')
             axs[0, 1].fill_between(sp.π_grid, 0, w_bar, color='blue', alpha=0.15)
             axs[0, 1].fill_between(sp.π_grid, w_bar, sp.w_max, color='green',
         ↪alpha=0.15)
             axs[0, 1].text(0.5, 0.6, 'reject')
```

```
        axs[0, 1].text(0.7, 0.9, 'accept')

        W = np.arange(0.01, 0.99, 0.08)
        Π = np.arange(0.01, 0.99, 0.08)

        ΔW = np.zeros((len(W), len(Π)))
        ΔΠ = np.empty((len(W), len(Π)))
        for i, w in enumerate(W):
            for j, π in enumerate(Π):
                lw = l(w)
                ΔΠ[i, j] = π * (lw / (π * lw + 1 - π) - 1)

        q = axs[0, 1].quiver(Π, W, ΔΠ, ΔW, scale=2, color='r', alpha=0.8)

        axs[0, 1].hlines(roots, 0., 1., linestyle="--")
        axs[0, 1].set(xlabel='$\pi$', ylabel='$w$')
        axs[0, 1].grid()

        axs[1, 0].plot(f(x_grid), x_grid, label='$f$', lw=2)
        axs[1, 0].plot(g(x_grid), x_grid, label='$g$', lw=2)
        axs[1, 0].vlines(1., 0., 1., linestyle="--")
        axs[1, 0].hlines(roots, 0., 2., linestyle="--")
        axs[1, 0].legend(loc=4)
        axs[1, 0].set(xlabel='$f(w), g(w)$', ylabel='$w$')

        axs[1, 1].plot(sp.π_grid, 1 - beta.cdf(w_bar, F_a, F_b), label='$f$')
        axs[1, 1].plot(sp.π_grid, 1 - beta.cdf(w_bar, G_a, G_b), label='$g$')
        axs[1, 1].set_ylim([0., 1.])
        axs[1, 1].grid()
        axs[1, 1].legend(loc=4)
        axs[1, 1].set(xlabel='$\pi$', ylabel='$\mathbb{P}\{w > \overline{w}(\pi)\}$')

        plt.show()

        # part 2: simulate empirical cumulative distribution
        accept_t, accept_π = empirical_dist(F_a, F_b, G_a, G_b, w_bar, π_grid)
        N = accept_t.shape[1]

        cfq_t_F = cumfreq(accept_t[0, :], numbins=100)
        cfq_π_F = cumfreq(accept_π[0, :], numbins=100)

        cfq_t_G = cumfreq(accept_t[1, :], numbins=100)
        cfq_π_G = cumfreq(accept_π[1, :], numbins=100)

        fig, axs = plt.subplots(2, 1, figsize=(12, 9))

        axs[0].plot(cumfreq_x(cfq_t_F), cfq_t_F.cumcount/N, label="f generates")
        axs[0].plot(cumfreq_x(cfq_t_G), cfq_t_G.cumcount/N, label="g generates")
        axs[0].grid(linestyle='--')
        axs[0].legend(loc=4)
        axs[0].title.set_text('CDF of duration of unemployment')
        axs[0].set(xlabel='time', ylabel='Prob(time)')

        axs[1].plot(cumfreq_x(cfq_π_F), cfq_π_F.cumcount/N, label="f generates")
```

```
        axs[1].plot(cumfreq_x(cfq_π_G), cfq_π_G.cumcount/N, label="g
↪generates")
        axs[1].grid(linestyle='--')
        axs[1].legend(loc=4)
        axs[1].title.set_text('CDF of π at time worker accepts wage and leaves
  unemployment')
        axs[1].set(xlabel='π', ylabel='Prob(π)')

        plt.show()
```

We now provide some examples that provide insights about how the model works.

42.13 Examples

42.13.1 Example 1 (Baseline)

$F \sim$ Beta(1, 1), $G \sim$ Beta(3, 1.2), c=0.3.

In the graphs below, the red arrows in the upper right figure show how π_t is updated in response to the new information w_t.

Recall the following formula from this lecture

$$\frac{\pi_{t+1}}{\pi_t} = \frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + \left(1 - \pi_t\right)} \begin{cases} > 1 & \text{if } l\left(w_{t+1}\right) > 1 \\ \leq 1 & \text{if } l\left(w_{t+1}\right) \leq 1 \end{cases}$$

The formula implies that the direction of motion of π_t is determined by the relationship between $l(w_t)$ and 1.

The magnitude is small if

- $l(w)$ is close to 1, which means the new w is not very informative for distinguishing two distributions,
- π_{t-1} is close to either 0 or 1, which means the priori is strong.

Will an unemployed worker accept an offer earlier or not, when the actual ruling distribution is g instead of f?

Two countervailing effects are at work.

- if f generates successive wage offers, then w is more likely to be low, but π is moving up toward to 1, which lowers the reservation wage, i.e., the worker becomes less selective the longer he or she remains unemployed.
- if g generates wage offers, then w is more likely to be high, but π is moving downward toward 0, increasing the reservation wage, i.e., the worker becomes more selective the longer he or she remains unemployed.

Quantitatively, the lower right figure sheds light on which effect dominates in this example.

It shows the probability that a previously unemployed worker accepts an offer at different values of π when f or g generates wage offers.

That graph shows that for the particular f and g in this example, the worker is always more likely to accept an offer when f generates the data even when π is close to zero so that the worker believes the true distribution is g and therefore is relatively more selective.

The empirical cumulative distribution of the duration of unemployment verifies our conjecture.

```
In [15]: job_search_example()
```

42.13.2 Example 2

$F \sim$ Beta(1, 1), $G \sim$ Beta(1.2, 1.2), c=0.3.

Now G has the same mean as F with a smaller variance.

Since the unemployment compensation c serves as a lower bound for bad wage offers, G is now an "inferior" distribution to F.

Consequently, we observe that the optimal policy $\overline{w}(\pi)$ is increasing in π.

```
In [16]: job_search_example(1, 1, 1.2, 1.2, 0.3)
```

$l(w) = f(w)/g(w)$
l
accept
reject
w
π
$f(w), g(w)$
f
g
$\mathbb{P}\{w > \overline{w}(\pi)\}$

CDF of duration of unemployment
Prob(time)
time
f generates
g generates

CDF of π at time worker accepts wage and leaves unemployment
Prob(π)
π
f generates
g generates

42.13.3 Example 3

$F \sim$ Beta(1, 1), $G \sim$ Beta(2, 2), c=0.3.

If the variance of G is smaller, we observe in the result that G is even more "inferior" and the slope of $\overline{w}(\pi)$ is larger.

```
In [17]: job_search_example(1, 1, 2, 2, 0.3)
```

42.13.4 Example 4

$F \sim$ Beta(1, 1), $G \sim$ Beta(3, 1.2), and c=0.8.

In this example, we keep the parameters of beta distributions to be the same with the baseline case but increase the unemployment compensation c.

Comparing outcomes to the baseline case (example 1) in which unemployment compensation if low (c=0.3), now the worker can afford a longer learning period.

As a result, the worker tends to accept wage offers much later.

Furthermore, at the time of accepting employment, the belief π is closer to either 0 or 1.

That means that the worker has a better idea about what the true distribution is when he eventually chooses to accept a wage offer.

```
In [18]: job_search_example(1, 1, 3, 1.2, c=0.8)
```

w
l
l(w) = f(w)/g(w)
accept
reject
π
f
g
f(w), g(w)
P{w > w̄(π)}

CDF of duration of unemployment
Prob(time)
time
f generates
g generates

CDF of π at time worker accepts wage and leaves unemployment
Prob(π)
π
f generates
g generates

42.13.5 Example 5

$F \sim$ Beta(1, 1), $G \sim$ Beta(3, 1.2), and c=0.1.

As expected, a smaller c makes an unemployed worker accept wage offers earlier after having acquired less information about the wage distribution.

```
In [19]: job_search_example(1, 1, 3, 1.2, c=0.1)
```

CDF of duration of unemployment
Prob(time)
time
f generates
g generates
CDF of π at time worker accepts wage and leaves unemployment
Prob(π)
π
f generates
g generates

Chapter 43

Job Search VI: Modeling Career Choice

43.1 Contents

- Overview 43.2
- Model 43.3
- Implementation 43.4
- Exercises 43.5
- Solutions 43.6

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

```
In [1]: !pip install --upgrade quantecon
```

43.2 Overview

Next, we study a computational problem concerning career and job choices.

The model is originally due to Derek Neal [121].

This exposition draws on the presentation in [108], section 6.5.

We begin with some imports:

```
In [2]: import numpy as np
        import quantecon as qe
        import matplotlib.pyplot as plt
        %matplotlib inline
        from numba import njit, prange
        from quantecon.distributions import BetaBinomial
        from scipy.special import binom, beta
        from mpl_toolkits.mplot3d.axes3d import Axes3D
        from matplotlib import cm
```

43.2.1 Model Features

- Career and job within career both chosen to maximize expected discounted wage flow.

- Infinite horizon dynamic programming with two state variables.

43.3 Model

In what follows we distinguish between a career and a job, where

- a *career* is understood to be a general field encompassing many possible jobs, and
- a *job* is understood to be a position with a particular firm

For workers, wages can be decomposed into the contribution of job and career

- $w_t = \theta_t + \epsilon_t$, where
  - θ_t is the contribution of career at time t
  - ϵ_t is the contribution of the job at time t

At the start of time t, a worker has the following options

- retain a current (career, job) pair (θ_t, ϵ_t) — referred to hereafter as "stay put"
- retain a current career θ_t but redraw a job ϵ_t — referred to hereafter as "new job"
- redraw both a career θ_t and a job ϵ_t — referred to hereafter as "new life"

Draws of θ and ϵ are independent of each other and past values, with

- $\theta_t \sim F$
- $\epsilon_t \sim G$

Notice that the worker does not have the option to retain a job but redraw a career — starting a new career always requires starting a new job.

A young worker aims to maximize the expected sum of discounted wages

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t w_t \tag{1}$$

subject to the choice restrictions specified above.

Let $v(\theta, \epsilon)$ denote the value function, which is the maximum of (1) overall feasible (career, job) policies, given the initial state (θ, ϵ).

The value function obeys

$$v(\theta, \epsilon) = \max\{I, II, III\}$$

where

$$\begin{aligned} I &= \theta + \epsilon + \beta v(\theta, \epsilon) \\ II &= \theta + \int \epsilon' G(d\epsilon') + \beta \int v(\theta, \epsilon') G(d\epsilon') \\ III &= \int \theta' F(d\theta') + \int \epsilon' G(d\epsilon') + \beta \int \int v(\theta', \epsilon') G(d\epsilon') F(d\theta') \end{aligned} \tag{2}$$

Evidently I, II and III correspond to "stay put", "new job" and "new life", respectively.

43.3.1 Parameterization

As in [108], section 6.5, we will focus on a discrete version of the model, parameterized as follows:

- both θ and ϵ take values in the set `np.linspace(0, B, grid_size)` — an even grid of points between 0 and B inclusive
- `grid_size = 50`
- `B = 5`
- `β = 0.95`

The distributions F and G are discrete distributions generating draws from the grid points `np.linspace(0, B, grid_size)`.

A very useful family of discrete distributions is the Beta-binomial family, with probability mass function

$$p(k \,|\, n, a, b) = \binom{n}{k} \frac{B(k+a, n-k+b)}{B(a,b)}, \qquad k = 0, \dots, n$$

Interpretation:

- draw q from a Beta distribution with shape parameters (a, b)
- run n independent binary trials, each with success probability q
- $p(k \,|\, n, a, b)$ is the probability of k successes in these n trials

Nice properties:

- very flexible class of distributions, including uniform, symmetric unimodal, etc.
- only three parameters

Here's a figure showing the effect on the pmf of different shape parameters when $n = 50$.

In [3]:
```
def gen_probs(n, a, b):
    probs = np.zeros(n+1)
    for k in range(n+1):
        probs[k] = binom(n, k) * beta(k + a, n - k + b) / beta(a, b)
    return probs

n = 50
a_vals = [0.5, 1, 100]
b_vals = [0.5, 1, 100]
fig, ax = plt.subplots(figsize=(10, 6))
for a, b in zip(a_vals, b_vals):
    ab_label = f'$a = {a:.1f}$, $b = {b:.1f}$'
    ax.plot(list(range(0, n+1)), gen_probs(n, a, b), '-o', label=ab_label)
ax.legend()
plt.show()
```

43.4 Implementation

We will first create a class `CareerWorkerProblem` which will hold the default parameterizations of the model and an initial guess for the value function.

```
In [4]: class CareerWorkerProblem:

            def __init__(self,
                         B=5.0,          # Upper bound
                         β=0.95,         # Discount factor
                         grid_size=50,   # Grid size
                         F_a=1,
                         F_b=1,
                         G_a=1,
                         G_b=1):

                self.β, self.grid_size, self.B = β, grid_size, B

                self.θ = np.linspace(0, B, grid_size)     # Set of ϑ values
                self.ϵ = np.linspace(0, B, grid_size)     # Set of ϵ values

                self.F_probs = BetaBinomial(grid_size - 1, F_a, F_b).pdf()
                self.G_probs = BetaBinomial(grid_size - 1, G_a, G_b).pdf()
                self.F_mean = np.sum(self.θ * self.F_probs)
                self.G_mean = np.sum(self.ϵ * self.G_probs)

                # Store these parameters for str and repr methods
                self._F_a, self._F_b = F_a, F_b
                self._G_a, self._G_b = G_a, G_b
```

The following function takes an instance of `CareerWorkerProblem` and returns the corresponding Bellman operator T and the greedy policy function.

In this model, T is defined by $Tv(\theta, \epsilon) = \max\{I, II, III\}$, where I, II and III are as given in (2).

```
In [5]: def operator_factory(cw, parallel_flag=True):

            """
            Returns jitted versions of the Bellman operator and the
            greedy policy function

            cw is an instance of ``CareerWorkerProblem``
            """

            θ, ϵ, β = cw.θ, cw.ϵ, cw.β
            F_probs, G_probs = cw.F_probs, cw.G_probs
            F_mean, G_mean = cw.F_mean, cw.G_mean

            @njit(parallel=parallel_flag)
            def T(v):
                "The Bellman operator"

                v_new = np.empty_like(v)

                for i in prange(len(v)):
                    for j in prange(len(v)):
                        v1 = θ[i] + ϵ[j] + β * v[i, j]                    # Stay put
                        v2 = θ[i] + G_mean + β * v[i, :] @ G_probs         # New job
                        v3 = G_mean + F_mean + β * F_probs @ v @ G_probs  # New life
                        v_new[i, j] = max(v1, v2, v3)

                return v_new

            @njit
            def get_greedy(v):
                "Computes the v-greedy policy"

                σ = np.empty(v.shape)

                for i in range(len(v)):
                    for j in range(len(v)):
                        v1 = θ[i] + ϵ[j] + β * v[i, j]
                        v2 = θ[i] + G_mean + β * v[i, :] @ G_probs
                        v3 = G_mean + F_mean + β * F_probs @ v @ G_probs
                        if v1 > max(v2, v3):
                            action = 1
                        elif v2 > max(v1, v3):
                            action = 2
                        else:
                            action = 3
                        σ[i, j] = action

                return σ

            return T, get_greedy
```

Lastly, `solve_model` will take an instance of `CareerWorkerProblem` and iterate using the Bellman operator to find the fixed point of the value function.

In [6]:
```
def solve_model(cw,
                use_parallel=True,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=25):

    T, _ = operator_factory(cw, parallel_flag=use_parallel)

    # Set up loop
    v = np.ones((cw.grid_size, cw.grid_size)) * 100  # Initial guess
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = T(v)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

    return v_new
```

Here's the solution to the model – an approximate value function

In [7]:
```
cw = CareerWorkerProblem()
T, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
tg, eg = np.meshgrid(cw.θ, cw.ϵ)
ax.plot_surface(tg,
                eg,
                v_star.T,
                cmap=cm.jet,
                alpha=0.5,
                linewidth=0.25)
ax.set(xlabel='θ', ylabel='ϵ', zlim=(150, 200))
ax.view_init(ax.elev, 225)
plt.show()
```

And here is the optimal policy

```
In [8]: fig, ax = plt.subplots(figsize=(6, 6))
        tg, eg = np.meshgrid(cw.θ, cw.ϵ)
        lvls = (0.5, 1.5, 2.5, 3.5)
        ax.contourf(tg, eg, greedy_star.T, levels=lvls, cmap=cm.winter, alpha=0.5)
        ax.contour(tg, eg, greedy_star.T, colors='k', levels=lvls, linewidths=2)
        ax.set(xlabel='θ', ylabel='ϵ')
        ax.text(1.8, 2.5, 'new life', fontsize=14)
        ax.text(4.5, 2.5, 'new job', fontsize=14, rotation='vertical')
        ax.text(4.0, 4.5, 'stay put', fontsize=14)
        plt.show()
```

Interpretation:

- If both job and career are poor or mediocre, the worker will experiment with a new job and new career.
- If career is sufficiently good, the worker will hold it and experiment with new jobs until a sufficiently good one is found.
- If both job and career are good, the worker will stay put.

Notice that the worker will always hold on to a sufficiently good career, but not necessarily hold on to even the best paying job.

The reason is that high lifetime wages require both variables to be large, and the worker cannot change careers without changing jobs.

- Sometimes a good job must be sacrificed in order to change to a better career.

43.5 Exercises

43.5.1 Exercise 1

Using the default parameterization in the class `CareerWorkerProblem`, generate and plot typical sample paths for θ and ϵ when the worker follows the optimal policy.

In particular, modulo randomness, reproduce the following figure (where the horizontal axis represents time)

Hint: To generate the draws from the distributions F and G, use `quantecon.random.draw()`.

43.5.2 Exercise 2

Let's now consider how long it takes for the worker to settle down to a permanent job, given a starting point of $(\theta, \epsilon) = (0, 0)$.

In other words, we want to study the distribution of the random variable

$$T^* := \text{the first point in time from which the worker's job no longer changes}$$

Evidently, the worker's job becomes permanent if and only if (θ_t, ϵ_t) enters the "stay put" region of (θ, ϵ) space.

Letting S denote this region, T^* can be expressed as the first passage time to S under the optimal policy:

$$T^* := \inf\{t \geq 0 \,|\, (\theta_t, \epsilon_t) \in S\}$$

Collect 25,000 draws of this random variable and compute the median (which should be about 7).

Repeat the exercise with $\beta = 0.99$ and interpret the change.

43.5.3 Exercise 3

Set the parameterization to `G_a = G_b = 100` and generate a new optimal policy figure – interpret.

43.6 Solutions

43.6.1 Exercise 1

Simulate job/career paths.

In reading the code, recall that `optimal_policy[i, j]` = policy at (θ_i, ϵ_j) = either 1, 2 or 3; meaning 'stay put', 'new job' and 'new life'.

In [9]:
```
F = np.cumsum(cw.F_probs)
G = np.cumsum(cw.G_probs)
v_star = solve_model(cw, verbose=False)
T, get_greedy = operator_factory(cw)
greedy_star = get_greedy(v_star)

def gen_path(optimal_policy, F, G, t=20):
    i = j = 0
    θ_index = []
    ϵ_index = []
    for t in range(t):
        if greedy_star[i, j] == 1:       # Stay put
            pass
        elif greedy_star[i, j] == 2:     # New job
            j = int(qe.random.draw(G))
        else:                            # New life
            i, j = int(qe.random.draw(F)), int(qe.random.draw(G))
        θ_index.append(i)
        ϵ_index.append(j)
    return cw.θ[θ_index], cw.ϵ[ϵ_index]


fig, axes = plt.subplots(2, 1, figsize=(10, 8))
for ax in axes:
    θ_path, ϵ_path = gen_path(greedy_star, F, G)
    ax.plot(ϵ_path, label='ϵ')
    ax.plot(θ_path, label='θ')
    ax.set_ylim(0, 6)

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

43.6.2 Exercise 2

The median for the original parameterization can be computed as follows

```
In [10]: cw = CareerWorkerProblem()
         F = np.cumsum(cw.F_probs)
         G = np.cumsum(cw.G_probs)
         T, get_greedy = operator_factory(cw)
         v_star = solve_model(cw, verbose=False)
         greedy_star = get_greedy(v_star)

         @njit
         def passage_time(optimal_policy, F, G):
             t = 0
             i = j = 0
             while True:
                 if optimal_policy[i, j] == 1:    # Stay put
                     return t
                 elif optimal_policy[i, j] == 2:  # New job
                     j = int(qe.random.draw(G))
                 else:                            # New life
                     i, j  = int(qe.random.draw(F)), int(qe.random.draw(G))
                 t += 1

         @njit(parallel=True)
         def median_time(optimal_policy, F, G, M=25000):
```

```
    samples = np.empty(M)
    for i in prange(M):
        samples[i] = passage_time(optimal_policy, F, G)
    return np.median(samples)

median_time(greedy_star, F, G)
```

Out[10]: 7.0

To compute the median with $\beta = 0.99$ instead of the default value $\beta = 0.95$, replace `cw = CareerWorkerProblem()` with `cw = CareerWorkerProblem(β=0.99)`.

The medians are subject to randomness but should be about 7 and 14 respectively.

Not surprisingly, more patient workers will wait longer to settle down to their final job.

43.6.3 Exercise 3

In [11]:
```
cw = CareerWorkerProblem(G_a=100, G_b=100)
T, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)

fig, ax = plt.subplots(figsize=(6, 6))
tg, eg = np.meshgrid(cw.θ, cw.ϵ)
lvls = (0.5, 1.5, 2.5, 3.5)
ax.contourf(tg, eg, greedy_star.T, levels=lvls, cmap=cm.winter, alpha=0.5)
ax.contour(tg, eg, greedy_star.T, colors='k', levels=lvls, linewidths=2)
ax.set(xlabel='θ', ylabel='ϵ')
ax.text(1.8, 2.5, 'new life', fontsize=14)
ax.text(4.5, 2.5, 'new job', fontsize=14, rotation='vertical')
ax.text(4.0, 4.5, 'stay put', fontsize=14)
plt.show()
```

In the new figure, you see that the region for which the worker stays put has grown because the distribution for ϵ has become more concentrated around the mean, making high-paying jobs less realistic.

Chapter 44

Job Search VII: On-the-Job Search

44.1 Contents

- Overview 44.2
- Model 44.3
- Implementation 44.4
- Solving for Policies 44.5
- Exercises 44.6
- Solutions 44.7

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

```
In [1]: !pip install --upgrade quantecon
        !pip install interpolation
```

44.2 Overview

In this section, we solve a simple on-the-job search model

- based on [108], exercise 6.18, and [91]

Let's start with some imports:

```
In [2]: import numpy as np
        import scipy.stats as stats
        from interpolation import interp
        from numba import njit, prange
        import matplotlib.pyplot as plt
        %matplotlib inline
        from math import gamma
```

44.2.1 Model Features

- job-specific human capital accumulation combined with on-the-job search
- infinite-horizon dynamic programming with one state variable and two controls

44.3 Model

Let x_t denote the time-t job-specific human capital of a worker employed at a given firm and let w_t denote current wages.

Let $w_t = x_t(1 - s_t - \phi_t)$, where

- ϕ_t is investment in job-specific human capital for the current role and
- s_t is search effort, devoted to obtaining new offers from other firms.

For as long as the worker remains in the current job, evolution of $\{x_t\}$ is given by $x_{t+1} = g(x_t, \phi_t)$.

When search effort at t is s_t, the worker receives a new job offer with probability $\pi(s_t) \in [0, 1]$.

The value of the offer, measured in job-specific human capital, is u_{t+1}, where $\{u_t\}$ is IID with common distribution f.

The worker can reject the current offer and continue with existing job.

Hence $x_{t+1} = u_{t+1}$ if he/she accepts and $x_{t+1} = g(x_t, \phi_t)$ otherwise.

Let $b_{t+1} \in \{0, 1\}$ be a binary random variable, where $b_{t+1} = 1$ indicates that the worker receives an offer at the end of time t.

We can write

$$x_{t+1} = (1 - b_{t+1})g(x_t, \phi_t) + b_{t+1} \max\{g(x_t, \phi_t), u_{t+1}\} \tag{1}$$

Agent's objective: maximize expected discounted sum of wages via controls $\{s_t\}$ and $\{\phi_t\}$.

Taking the expectation of $v(x_{t+1})$ and using (1), the Bellman equation for this problem can be written as

$$v(x) = \max_{s+\phi \leq 1} \left\{ x(1 - s - \phi) + \beta(1 - \pi(s))v[g(x, \phi)] + \beta\pi(s) \int v[g(x, \phi) \vee u] f(du) \right\} \tag{2}$$

Here nonnegativity of s and ϕ is understood, while $a \vee b := \max\{a, b\}$.

44.3.1 Parameterization

In the implementation below, we will focus on the parameterization

$$g(x, \phi) = A(x\phi)^\alpha, \quad \pi(s) = \sqrt{s} \quad \text{and} \quad f = \text{Beta}(2, 2)$$

with default parameter values

- $A = 1.4$
- $\alpha = 0.6$
- $\beta = 0.96$

The $\text{Beta}(2, 2)$ distribution is supported on $(0, 1)$ - it has a unimodal, symmetric density peaked at 0.5.

44.3.2 Back-of-the-Envelope Calculations

Before we solve the model, let's make some quick calculations that provide intuition on what the solution should look like.

To begin, observe that the worker has two instruments to build capital and hence wages:

1. invest in capital specific to the current job via ϕ
2. search for a new job with better job-specific capital match via s

Since wages are $x(1-s-\phi)$, marginal cost of investment via either ϕ or s is identical.

Our risk-neutral worker should focus on whatever instrument has the highest expected return.

The relative expected return will depend on x.

For example, suppose first that $x = 0.05$

- If $s = 1$ and $\phi = 0$, then since $g(x, \phi) = 0$, taking expectations of (1) gives expected next period capital equal to $\pi(s)\mathbb{E}u = \mathbb{E}u = 0.5$.
- If $s = 0$ and $\phi = 1$, then next period capital is $g(x, \phi) = g(0.05, 1) \approx 0.23$.

Both rates of return are good, but the return from search is better.

Next, suppose that $x = 0.4$

- If $s = 1$ and $\phi = 0$, then expected next period capital is again 0.5
- If $s = 0$ and $\phi = 1$, then $g(x, \phi) = g(0.4, 1) \approx 0.8$

Return from investment via ϕ dominates expected return from search.

Combining these observations gives us two informal predictions:

1. At any given state x, the two controls ϕ and s will function primarily as substitutes — worker will focus on whichever instrument has the higher expected return.
2. For sufficiently small x, search will be preferable to investment in job-specific human capital. For larger x, the reverse will be true.

Now let's turn to implementation, and see if we can match our predictions.

44.4 Implementation

We will set up a class `JVWorker` that holds the parameters of the model described above

```
In [3]: class JVWorker:
            r"""
            A Jovanovic-type model of employment with on-the-job search.

            """

            def __init__(self,
                         A=1.4,
                         α=0.6,
                         β=0.96,          # Discount factor
```

```
                 π=np.sqrt,       # Search effort function
                 a=2,             # Parameter of f
                 b=2,             # Parameter of f
                 grid_size=50,
                 mc_size=100,
                 □=1e-4):

        self.A, self.α, self.β, self.π = A, α, β, π
        self.mc_size, self.□ = mc_size, □

        self.g = njit(lambda x, ϕ: A * (x * ϕ)**α)    # Transition function
        self.f_rvs = np.random.beta(a, b, mc_size)

        # Max of grid is the max of a large quantile value for f and the
        # fixed point y = g(y, 1)
        □ = 1e-4
        grid_max = max(A**(1 / (1 - α)), stats.beta(a, b).ppf(1 - □))

        # Human capital
        self.x_grid = np.linspace(□, grid_max, grid_size)
```

The function `operator_factory` takes an instance of this class and returns a jitted version of the Bellman operator `T`, ie.

$$Tv(x) = \max_{s+\phi \leq 1} w(s, \phi)$$

where

$$w(s, \phi) := x(1 - s - \phi) + \beta(1 - \pi(s))v[g(x, \phi)] + \beta\pi(s) \int v[g(x, \phi) \vee u] f(du) \tag{3}$$

When we represent v, it will be with a NumPy array `v` giving values on grid `x_grid`.

But to evaluate the right-hand side of (3), we need a function, so we replace the arrays `v` and `x_grid` with a function `v_func` that gives linear interpolation of `v` on `x_grid`.

Inside the `for` loop, for each `x` in the grid over the state space, we set up the function $w(z) = w(s, \phi)$ defined in (3).

The function is maximized over all feasible (s, ϕ) pairs.

Another function, `get_greedy` returns the optimal choice of s and ϕ at each x, given a value function.

```
In [4]: def operator_factory(jv, parallel_flag=True):

            """
            Returns a jitted version of the Bellman operator T

            jv is an instance of JVWorker

            """

            π, β = jv.π, jv.β
            x_grid, □, mc_size = jv.x_grid, jv.□, jv.mc_size
            f_rvs, g = jv.f_rvs, jv.g
```

```
@njit
def objective(z, x, v):
    s, ϕ = z
    v_func = lambda x: interp(x_grid, v, x)

    integral = 0
    for m in range(mc_size):
        u = f_rvs[m]
        integral += v_func(max(g(x, ϕ), u))
    integral = integral / mc_size

    q = π(s) * integral + (1 - π(s)) * v_func(g(x, ϕ))
    return x * (1 - ϕ - s) + β * q

@njit(parallel=parallel_flag)
def T(v):
    """
    The Bellman operator
    """

    v_new = np.empty_like(v)
    for i in prange(len(x_grid)):
        x = x_grid[i]

        # Search on a grid
        search_grid = np.linspace(□, 1, 15)
        max_val = -1
        for s in search_grid:
            for ϕ in search_grid:
             current_val = objective((s, ϕ), x, v) if s + ϕ <= 1 else -1
                if current_val > max_val:
                    max_val = current_val
        v_new[i] = max_val

    return v_new

@njit
def get_greedy(v):
    """
    Computes the v-greedy policy of a given function v
    """
    s_policy, ϕ_policy = np.empty_like(v), np.empty_like(v)

    for i in range(len(x_grid)):
        x = x_grid[i]
        # Search on a grid
        search_grid = np.linspace(□, 1, 15)
        max_val = -1
        for s in search_grid:
            for ϕ in search_grid:
             current_val = objective((s, ϕ), x, v) if s + ϕ <= 1 else -1
                if current_val > max_val:
                    max_val = current_val
                    max_s, max_ϕ = s, ϕ
                    s_policy[i], ϕ_policy[i] = max_s, max_ϕ
    return s_policy, ϕ_policy
```

```
    return T, get_greedy
```

To solve the model, we will write a function that uses the Bellman operator and iterates to find a fixed point.

```
In [5]: def solve_model(jv,
                        use_parallel=True,
                        tol=1e-4,
                        max_iter=1000,
                        verbose=True,
                        print_skip=25):

            """
            Solves the model by value function iteration

            * jv is an instance of JVWorker

            """

            T, _ = operator_factory(jv, parallel_flag=use_parallel)

            # Set up loop
            v = jv.x_grid * 0.5  # Initial condition
            i = 0
            error = tol + 1

            while i < max_iter and error > tol:
                v_new = T(v)
                error = np.max(np.abs(v - v_new))
                i += 1
                if verbose and i % print_skip == 0:
                    print(f"Error at iteration {i} is {error}.")
                v = v_new

            if i == max_iter:
                print("Failed to converge!")

            if verbose and i < max_iter:
                print(f"\nConverged in {i} iterations.")

            return v_new
```

44.5 Solving for Policies

Let's generate the optimal policies and see what they look like.

```
In [6]: jv = JVWorker()
        T, get_greedy = operator_factory(jv)
        v_star = solve_model(jv)
        s_star, ϕ_star = get_greedy(v_star)
```

```
    Error at iteration 25 is 0.15111146305839807.
Error at iteration 50 is 0.05446007514020579.
Error at iteration 75 is 0.01962723228037433.
```

```
Error at iteration 100 is 0.007073590074865521.
Error at iteration 125 is 0.0025492986393889083.
Error at iteration 150 is 0.0009187588599299801.
Error at iteration 175 is 0.00033111767670845893.
Error at iteration 200 is 0.00011933372358008398.

Converged in 205 iterations.
```

Here are the plots:

```
In [7]: plots = [s_star, ϕ_star, v_star]
        titles = ["s policy", "ϕ policy",  "value function"]

        fig, axes = plt.subplots(3, 1, figsize=(12, 12))

        for ax, plot, title in zip(axes, plots, titles):
            ax.plot(jv.x_grid, plot)
            ax.set(title=title)
            ax.grid()

        axes[-1].set_xlabel("x")
        plt.show()
```

The horizontal axis is the state x, while the vertical axis gives $s(x)$ and $\phi(x)$.

Overall, the policies match well with our predictions from above

- Worker switches from one investment strategy to the other depending on relative return.
- For low values of x, the best option is to search for a new job.
- Once x is larger, worker does better by investing in human capital specific to the current position.

44.6 Exercises

44.6.1 Exercise 1

Let's look at the dynamics for the state process $\{x_t\}$ associated with these policies.

The dynamics are given by (1) when ϕ_t and s_t are chosen according to the optimal policies, and $\mathbb{P}\{b_{t+1} = 1\} = \pi(s_t)$.

Since the dynamics are random, analysis is a bit subtle.

One way to do it is to plot, for each x in a relatively fine grid called `plot_grid`, a large number K of realizations of x_{t+1} given $x_t = x$.

Plot this with one dot for each realization, in the form of a 45 degree diagram, setting

```
jv = JVWorker(grid_size=25, mc_size=50)
plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = np.linspace(0, plot_grid_max, plot_grid_size)
fig, ax = plt.subplots()
ax.set_xlim(0, plot_grid_max)
ax.set_ylim(0, plot_grid_max)
```

By examining the plot, argue that under the optimal policies, the state x_t will converge to a constant value $\bar{x}$ close to unity.

Argue that at the steady state, $s_t \approx 0$ and $\phi_t \approx 0.6$.

44.6.2 Exercise 2

In the preceding exercise, we found that s_t converges to zero and ϕ_t converges to about 0.6.

Since these results were calculated at a value of β close to one, let's compare them to the best choice for an *infinitely* patient worker.

Intuitively, an infinitely patient worker would like to maximize steady state wages, which are a function of steady state capital.

You can take it as given—it's certainly true—that the infinitely patient worker does not search in the long run (i.e., $s_t = 0$ for large t).

Thus, given ϕ, steady state capital is the positive fixed point $x^*(\phi)$ of the map $x \mapsto g(x, \phi)$.

Steady state wages can be written as $w^*(\phi) = x^*(\phi)(1 - \phi)$.

Graph $w^*(\phi)$ with respect to ϕ, and examine the best choice of ϕ.

Can you give a rough interpretation for the value that you see?

44.7 Solutions

44.7.1 Exercise 1

Here's code to produce the 45 degree diagram

```
In [8]: jv = JVWorker(grid_size=25, mc_size=50)
        π, g, f_rvs, x_grid = jv.π, jv.g, jv.f_rvs, jv.x_grid
        T, get_greedy = operator_factory(jv)
        v_star = solve_model(jv, verbose=False)
        s_policy, ϕ_policy = get_greedy(v_star)

        # Turn the policy function arrays into actual functions
        s = lambda y: interp(x_grid, s_policy, y)
        ϕ = lambda y: interp(x_grid, ϕ_policy, y)

        def h(x, b, u):
            return (1 - b) * g(x, ϕ(x)) + b * max(g(x, ϕ(x)), u)


        plot_grid_max, plot_grid_size = 1.2, 100
        plot_grid = np.linspace(0, plot_grid_max, plot_grid_size)
        fig, ax = plt.subplots(figsize=(8, 8))
        ticks = (0.25, 0.5, 0.75, 1.0)
        ax.set(xticks=ticks, yticks=ticks,
               xlim=(0, plot_grid_max),
               ylim=(0, plot_grid_max),
               xlabel='$x_t$', ylabel='$x_{t+1}$')

        ax.plot(plot_grid, plot_grid, 'k--', alpha=0.6)  # 45 degree line
        for x in plot_grid:
            for i in range(jv.mc_size):
                b = 1 if np.random.uniform(0, 1) < π(s(x)) else 0
                u = f_rvs[i]
                y = h(x, b, u)
                ax.plot(x, y, 'go', alpha=0.25)

        plt.show()
```

Looking at the dynamics, we can see that

- If x_t is below about 0.2 the dynamics are random, but $x_{t+1} > x_t$ is very likely.
- As x_t increases the dynamics become deterministic, and x_t converges to a steady state value close to 1.

Referring back to the figure here we see that $x_t \approx 1$ means that $s_t = s(x_t) \approx 0$ and $\phi_t = \phi(x_t) \approx 0.6$.

44.7.2 Exercise 2

The figure can be produced as follows

```
In [9]: jv = JVWorker()

        def xbar(ϕ):
            A, α = jv.A, jv.α
            return (A * ϕ**α)**(1 / (1 - α))

        ϕ_grid = np.linspace(0, 1, 100)
        fig, ax = plt.subplots(figsize=(9, 7))
        ax.set(xlabel='$\phi$')
```

```
ax.plot(ϕ_grid, [xbar(ϕ) * (1 - ϕ) for ϕ in ϕ_grid], label='$w^*(\phi)$')
ax.legend()

plt.show()
```

Observe that the maximizer is around 0.6.

This is similar to the long-run value for ϕ obtained in exercise 1.

Hence the behavior of the infinitely patent worker is similar to that of the worker with $\beta = 0.96$.

This seems reasonable and helps us confirm that our dynamic programming solutions are probably correct.

Chapter 45

A Problem that Stumped Milton Friedman

(and that Abraham Wald solved by inventing sequential analysis)

45.1 Contents

- Overview 45.2
- Origin of the Problem 45.3
- A Dynamic Programming Approach 45.4
- Implementation 45.5
- Analysis 45.6
- Comparison with Neyman-Pearson Formulation 45.7

Co-authors: Chase Coleman

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

```
In [1]: !pip install --upgrade quantecon
        !pip install interpolation
```

45.2 Overview

This lecture describes a statistical decision problem encountered by Milton Friedman and W. Allen Wallis during World War II when they were analysts at the U.S. Government's Statistical Research Group at Columbia University.

This problem led Abraham Wald [162] to formulate **sequential analysis**, an approach to statistical decision problems intimately related to dynamic programming.

In this lecture, we apply dynamic programming algorithms to Friedman and Wallis and Wald's problem.

Key ideas in play will be:

- Bayes' Law
- Dynamic programming
- Type I and type II statistical errors

  - a type I error occurs when you reject a null hypothesis that is true
  - a type II error is when you accept a null hypothesis that is false
- Abraham Wald's **sequential probability ratio test**
- The **power** of a statistical test
- The **critical region** of a statistical test
- A **uniformly most powerful test**

We'll begin with some imports:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        from numba import njit, prange, vectorize
        from interpolation import interp
        from math import gamma
```

45.3 Origin of the Problem

On pages 137-139 of his 1998 book *Two Lucky People* with Rose Friedman [57], Milton Friedman described a problem presented to him and Allen Wallis during World War II, when they worked at the US Government's Statistical Research Group at Columbia University.

Let's listen to Milton Friedman tell us what happened

> In order to understand the story, it is necessary to have an idea of a simple statistical problem, and of the standard procedure for dealing with it. The actual problem out of which sequential analysis grew will serve. The Navy has two alternative designs (say A and B) for a projectile. It wants to determine which is superior. To do so it undertakes a series of paired firings. On each round, it assigns the value 1 or 0 to A accordingly as its performance is superior or inferior to that of B and conversely 0 or 1 to B. The Navy asks the statistician how to conduct the test and how to analyze the results.
>
> The standard statistical answer was to specify a number of firings (say 1,000) and a pair of percentages (e.g., 53% and 47%) and tell the client that if A receives a 1 in more than 53% of the firings, it can be regarded as superior; if it receives a 1 in fewer than 47%, B can be regarded as superior; if the percentage is between 47% and 53%, neither can be so regarded.
>
> When Allen Wallis was discussing such a problem with (Navy) Captain Garret L. Schyler, the captain objected that such a test, to quote from Allen's account, may prove wasteful. If a wise and seasoned ordnance officer like Schyler were on the premises, he would see after the first few thousand or even few hundred [rounds] that the experiment need not be completed either because the new method is obviously inferior or because it is obviously superior beyond what was hoped for ....

Friedman and Wallis struggled with the problem but, after realizing that they were not able to solve it, described the problem to Abraham Wald.

That started Wald on the path that led him to *Sequential Analysis* [162].

We'll formulate the problem using dynamic programming.

45.4 A Dynamic Programming Approach

The following presentation of the problem closely follows Dmitri Berskekas's treatment in **Dynamic Programming and Stochastic Control** [20].

A decision-maker observes a sequence of draws of a random variable z.

He (or she) wants to know which of two probability distributions f_0 or f_1 governs z.

Conditional on knowing that successive observations are drawn from distribution f_0, the sequence of random variables is independently and identically distributed (IID).

Conditional on knowing that successive observations are drawn from distribution f_1, the sequence of random variables is also independently and identically distributed (IID).

But the observer does not know which of the two distributions generated the sequence.

For reasons explained Exchangeability and Bayesian Updating, this means that the sequence is not IID and that the observer has something to learn, even though he knows both f_0 and f_1.

After a number of draws, also to be determined, he makes a decision about which of the distributions is generating the draws he observes.

He starts with prior

$$\pi_{-1} = \mathbb{P}\{f = f_0 \mid \text{ no observations}\} \in (0,1)$$

After observing $k+1$ observations $z_k, z_{k-1}, \dots, z_0$, he updates this value to

$$\pi_k = \mathbb{P}\{f = f_0 \mid z_k, z_{k-1}, \dots, z_0\}$$

which is calculated recursively by applying Bayes' law:

$$\pi_{k+1} = \frac{\pi_k f_0(z_{k+1})}{\pi_k f_0(z_{k+1}) + (1-\pi_k) f_1(z_{k+1})}, \quad k = -1, 0, 1, \dots$$

After observing $z_k, z_{k-1}, \dots, z_0$, the decision-maker believes that z_{k+1} has probability distribution

$$f_{\pi_k}(v) = \pi_k f_0(v) + (1-\pi_k) f_1(v)$$

This is a mixture of distributions f_0 and f_1, with the weight on f_0 being the posterior probability that $f = f_0$ Section **??**.

To help illustrate this kind of distribution, let's inspect some mixtures of beta distributions.

The density of a beta probability distribution with parameters a and b is

$$f(z; a, b) = \frac{\Gamma(a+b) z^{a-1} (1-z)^{b-1}}{\Gamma(a)\Gamma(b)} \quad \text{where} \quad \Gamma(t) := \int_0^\infty x^{t-1} e^{-x} dx$$

The next figure shows two beta distributions in the top panel.

The bottom panel presents mixtures of these distributions, with various mixing probabilities π_k

In [3]:
```
def beta_function_factory(a, b):

    @vectorize
    def p(x):
        r = gamma(a + b) / (gamma(a) * gamma(b))
        return r * x**(a-1) * (1 - x)**(b-1)

    @njit
    def p_rvs():
        return np.random.beta(a, b)

    return p, p_rvs


f0, _ = beta_function_factory(1, 1)
f1, _ = beta_function_factory(9, 9)
grid = np.linspace(0, 1, 50)

fig, axes = plt.subplots(2, figsize=(10, 8))

axes[0].set_title("Original Distributions")
axes[0].plot(grid, f0(grid), lw=2, label="$f_0$")
axes[0].plot(grid, f1(grid), lw=2, label="$f_1$")

axes[1].set_title("Mixtures")
for π in 0.25, 0.5, 0.75:
    y = π * f0(grid) + (1 - π) * f1(grid)
    axes[1].plot(y, lw=2, label=f"$\pi_k$ = {π}")

for ax in axes:
    ax.legend()
    ax.set(xlabel="$z$ values", ylabel="probability of $z_k$")

plt.tight_layout()
plt.show()
```

45.4.1 Losses and Costs

After observing $z_k, z_{k-1}, \ldots, z_0$, the decision-maker chooses among three distinct actions:

- He decides that $f = f_0$ and draws no more z's
- He decides that $f = f_1$ and draws no more z's
- He postpones deciding now and instead chooses to draw a z_{k+1}

Associated with these three actions, the decision-maker can suffer three kinds of losses:

- A loss L_0 if he decides $f = f_0$ when actually $f = f_1$
- A loss L_1 if he decides $f = f_1$ when actually $f = f_0$
- A cost c if he postpones deciding and chooses instead to draw another z

45.4.2 Digression on Type I and Type II Errors

If we regard $f = f_0$ as a null hypothesis and $f = f_1$ as an alternative hypothesis, then L_1 and L_0 are losses associated with two types of statistical errors

- a type I error is an incorrect rejection of a true null hypothesis (a "false positive")
- a type II error is a failure to reject a false null hypothesis (a "false negative")

So when we treat $f = f_0$ as the null hypothesis

- We can think of L_1 as the loss associated with a type I error.
- We can think of L_0 as the loss associated with a type II error.

45.4.3 Intuition

Let's try to guess what an optimal decision rule might look like before we go further.

Suppose at some given point in time that π is close to 1.

Then our prior beliefs and the evidence so far point strongly to $f = f_0$.

If, on the other hand, π is close to 0, then $f = f_1$ is strongly favored.

Finally, if π is in the middle of the interval $[0, 1]$, then we have little information in either direction.

This reasoning suggests a decision rule such as the one shown in the figure

As we'll see, this is indeed the correct form of the decision rule.

The key problem is to determine the threshold values α, β, which will depend on the parameters listed above.

You might like to pause at this point and try to predict the impact of a parameter such as c or L_0 on α or β.

45.4.4 A Bellman Equation

Let $J(\pi)$ be the total loss for a decision-maker with current belief π who chooses optimally.

With some thought, you will agree that J should satisfy the Bellman equation

$$J(\pi) = \min \{(1-\pi)L_0,\ \pi L_1,\ c + \mathbb{E}[J(\pi')]\} \tag{1}$$

where π' is the random variable defined by

$$\pi' = \kappa(z', \pi) = \frac{\pi f_0(z')}{\pi f_0(z') + (1-\pi) f_1(z')}$$

when π is fixed and z' is drawn from the current best guess, which is the distribution f defined by

$$f_\pi(v) = \pi f_0(v) + (1-\pi) f_1(v)$$

In the Bellman equation, minimization is over three actions:

1. Accept the hypothesis that $f = f_0$
2. Accept the hypothesis that $f = f_1$
3. Postpone deciding and draw again

We can represent the Bellman equation as

$$J(\pi) = \min\{(1-\pi)L_0,\ \pi L_1,\ h(\pi)\} \tag{2}$$

where $\pi \in [0,1]$ and

- $(1-\pi)L_0$ is the expected loss associated with accepting f_0 (i.e., the cost of making a type II error).
- πL_1 is the expected loss associated with accepting f_1 (i.e., the cost of making a type I error).
- $h(\pi) := c + \mathbb{E}[J(\pi')]$ the continuation value; i.e., the expected cost associated with drawing one more z.

The optimal decision rule is characterized by two numbers $\alpha, \beta \in (0,1) \times (0,1)$ that satisfy

$$(1-\pi)L_0 < \min\{\pi L_1, c + \mathbb{E}[J(\pi')]\} \text{ if } \pi \geq \alpha$$

and

$$\pi L_1 < \min\{(1-\pi)L_0, c + \mathbb{E}[J(\pi')]\} \text{ if } \pi \leq \beta$$

The optimal decision rule is then

$$\begin{aligned} \text{accept } f = f_0 \text{ if } \pi \geq \alpha \\ \text{accept } f = f_1 \text{ if } \pi \leq \beta \\ \text{draw another } z \text{ if } \beta \leq \pi \leq \alpha \end{aligned}$$

Our aim is to compute the value function J, and from it the associated cutoffs α and β.

To make our computations simpler, using (2), we can write the continuation value $h(\pi)$ as

$$\begin{aligned} h(\pi) &= c + \mathbb{E}[J(\pi')] \\ &= c + \mathbb{E}_{\pi'} \min\{(1-\pi')L_0, \pi' L_1, h(\pi')\} \\ &= c + \int \min\{(1-\kappa(z',\pi))L_0, \kappa(z',\pi)L_1, h(\kappa(z',\pi))\} f_\pi(z') dz' \end{aligned} \tag{3}$$

The equality

$$h(\pi) = c + \int \min\{(1-\kappa(z',\pi))L_0, \kappa(z',\pi)L_1, h(\kappa(z',\pi))\} f_\pi(z') dz' \tag{4}$$

can be understood as a functional equation, where h is the unknown.

Using the functional equation, (4), for the continuation value, we can back out optimal choices using the RHS of (2).

This functional equation can be solved by taking an initial guess and iterating to find the fixed point.

In other words, we iterate with an operator Q, where

$$Qh(\pi) = c + \int \min\{(1-\kappa(z',\pi))L_0, \kappa(z',\pi)L_1, h(\kappa(z',\pi))\} f_\pi(z') dz'$$

45.5 Implementation

First, we will construct a class to store the parameters of the model

In [4]:
```
class WaldFriedman:

    def __init__(self,
                 c=1.25,          # Cost of another draw
                 a0=1,
                 b0=1,
                 a1=3,
                 b1=1.2,
                 L0=25,           # Cost of selecting f0 when f1 is true
                 L1=25,           # Cost of selecting f1 when f0 is true
                 π_grid_size=200,
                 mc_size=1000):

        self.c, self.π_grid_size = c, π_grid_size
        self.L0, self.L1 = L0, L1
        self.π_grid = np.linspace(0, 1, π_grid_size)
        self.mc_size = mc_size

        # Set up distributions
        self.f0, self.f0_rvs = beta_function_factory(a0, b0)
        self.f1, self.f1_rvs = beta_function_factory(a1, b1)

        self.z0 = np.random.beta(a0, b0, mc_size)
        self.z1 = np.random.beta(a1, b1, mc_size)
```

As in the optimal growth lecture, to approximate a continuous value function

- We iterate at a finite grid of possible values of π.
- When we evaluate $\mathbb{E}[J(\pi')]$ between grid points, we use linear interpolation.

The function `operator_factory` returns the operator `Q`

In [5]:
```
def operator_factory(wf, parallel_flag=True):

    """
    Returns a jitted version of the Q operator.

    * wf is an instance of the WaldFriedman class
    """

    c, π_grid = wf.c, wf.π_grid
    L0, L1 = wf.L0, wf.L1
    f0, f1 = wf.f0, wf.f1
    z0, z1 = wf.z0, wf.z1
    mc_size = wf.mc_size

    @njit
    def κ(z, π):
        """
        Updates π using Bayes' rule and the current observation z
        """
        π_f0, π_f1 = π * f0(z), (1 - π) * f1(z)
        π_new = π_f0 / (π_f0 + π_f1)
```

```
        return π_new

    @njit(parallel=parallel_flag)
    def Q(h):
        h_new = np.empty_like(π_grid)
        h_func = lambda p: interp(π_grid, h, p)

        for i in prange(len(π_grid)):
            π = π_grid[i]

            # Find the expected value of J by integrating over z
            integral_f0, integral_f1 = 0, 0
            for m in range(mc_size):
                π_0 = κ(z0[m], π)  # Draw z from f0 and update π
                integral_f0 += min((1 - π_0) * L0, π_0 * L1, h_func(π_0))

                π_1 = κ(z1[m], π)  # Draw z from f1 and update π
                integral_f1 += min((1 - π_1) * L0, π_1 * L1, h_func(π_1))

            integral = (π * integral_f0 + (1 - π) * integral_f1) / mc_size

            h_new[i] = c + integral

        return h_new

    return Q
```

To solve the model, we will iterate using `Q` to find the fixed point

```
In [6]: def solve_model(wf,
                        use_parallel=True,
                        tol=1e-4,
                        max_iter=1000,
                        verbose=True,
                        print_skip=25):

    """
    Compute the continuation value function

    * wf is an instance of WaldFriedman
    """

    Q = operator_factory(wf, parallel_flag=use_parallel)

    # Set up loop
    h = np.zeros(len(wf.π_grid))
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        h_new = Q(h)
        error = np.max(np.abs(h - h_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        h = h_new
```

```
    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

    return h_new
```

45.6 Analysis

Let's inspect the model's solutions.

We will be using the default parameterization with distributions like so

```
In [7]: wf = WaldFriedman()

        fig, ax = plt.subplots(figsize=(10, 6))
        ax.plot(wf.f0(wf.π_grid), label="$f_0$")
        ax.plot(wf.f1(wf.π_grid), label="$f_1$")
        ax.set(ylabel="probability of $z_k$", xlabel="$k$", title="Distributions")
        ax.legend()

        plt.show()
```

45.6.1 Value Function

To solve the model, we will call our `solve_model` function

In [8]:
```
h_star = solve_model(wf)    # Solve the model
```

```
    Error at iteration 25 is 8.554240983382755e-05.

Converged in 25 iterations.
```

We will also set up a function to compute the cutoffs α and β and plot these on our value function plot

In [9]:
```
def find_cutoff_rule(wf, h):

    """
    This function takes a continuation value function and returns the
    corresponding cutoffs of where you transition between continuing and
    choosing a specific model
    """

    π_grid = wf.π_grid
    L0, L1 = wf.L0, wf.L1

    # Evaluate cost at all points on grid for choosing a model
    payoff_f0 = (1 - π_grid) * L0
    payoff_f1 = π_grid * L1

    # The cutoff points can be found by differencing these costs with
    # The Bellman equation (J is always less than or equal to p_c_i)
    β = π_grid[np.searchsorted(
                              payoff_f1 - np.minimum(h, payoff_f0),
                              1e-10)
               - 1]
    α = π_grid[np.searchsorted(
                              np.minimum(h, payoff_f1) - payoff_f0,
                              1e-10)
               - 1]

    return (β, α)

β, α = find_cutoff_rule(wf, h_star)
cost_L0 = (1 - wf.π_grid) * wf.L0
cost_L1 = wf.π_grid * wf.L1

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(wf.π_grid, h_star, label='continuation value')
ax.plot(wf.π_grid, cost_L1, label='choose f1')
ax.plot(wf.π_grid, cost_L0, label='choose f0')
ax.plot(wf.π_grid,
        np.amin(np.column_stack([h_star, cost_L0, cost_L1]),axis=1),
        lw=15, alpha=0.1, color='b', label='minimum cost')

ax.annotate(r"$\beta$", xy=(β + 0.01, 0.5), fontsize=14)
ax.annotate(r"$\alpha$", xy=(α + 0.01, 0.5), fontsize=14)

plt.vlines(β, 0, β * wf.L0, linestyle="--")
plt.vlines(α, 0, (1 - α) * wf.L1, linestyle="--")
```

```
ax.set(xlim=(0, 1), ylim=(0, 0.5 * max(wf.L0, wf.L1)), ylabel="cost",
       xlabel="$\pi$", title="Value function")

plt.legend(borderpad=1.1)
plt.show()
```

The value function equals πL_1 for $\pi \leq \beta$, and $(1-\pi)L_0$ for $\pi \geq \alpha$.

The slopes of the two linear pieces of the value function are determined by L_1 and $-L_0$.

The value function is smooth in the interior region, where the posterior probability assigned to f_0 is in the indecisive region $\pi \in (\beta, \alpha)$.

The decision-maker continues to sample until the probability that he attaches to model f_0 falls below β or above α.

45.6.2 Simulations

The next figure shows the outcomes of 500 simulations of the decision process.

On the left is a histogram of the stopping times, which equal the number of draws of z_k required to make a decision.

The average number of draws is around 6.6.

On the right is the fraction of correct decisions at the stopping time.

In this case, the decision-maker is correct 80% of the time

```
In [10]: def simulate(wf, true_dist, h_star, π_0=0.5):

             """
             This function takes an initial condition and simulates until it
```

```
    stops (when a decision is made)
    """

    f0, f1 = wf.f0, wf.f1
    f0_rvs, f1_rvs = wf.f0_rvs, wf.f1_rvs
    π_grid = wf.π_grid

    def κ(z, π):
        """
        Updates π using Bayes' rule and the current observation z
        """
        π_f0, π_f1 = π * f0(z), (1 - π) * f1(z)
        π_new = π_f0 / (π_f0 + π_f1)

        return π_new

    if true_dist == "f0":
        f, f_rvs = wf.f0, wf.f0_rvs
    elif true_dist == "f1":
        f, f_rvs = wf.f1, wf.f1_rvs

    # Find cutoffs
    β, α = find_cutoff_rule(wf, h_star)

    # Initialize a couple of useful variables
    decision_made = False
    π = π_0
    t = 0

    while decision_made is False:
        # Maybe should specify which distribution is correct one so that
        # the draws come from the "right" distribution
        z = f_rvs()
        t = t + 1
        π = κ(z, π)
        if π < β:
            decision_made = True
            decision = 1
        elif π > α:
            decision_made = True
            decision = 0

    if true_dist == "f0":
        if decision == 0:
            correct = True
        else:
            correct = False

    elif true_dist == "f1":
        if decision == 1:
            correct = True
        else:
            correct = False

    return correct, π, t

def stopping_dist(wf, h_star, ndraws=250, true_dist="f0"):
```

```
    """
    Simulates repeatedly to get distributions of time needed to make a
    decision and how often they are correct
    """

    tdist = np.empty(ndraws, int)
    cdist = np.empty(ndraws, bool)

    for i in range(ndraws):
        correct, π, t = simulate(wf, true_dist, h_star)
        tdist[i] = t
        cdist[i] = correct

    return cdist, tdist

def simulation_plot(wf):
    h_star = solve_model(wf)
    ndraws = 500
    cdist, tdist = stopping_dist(wf, h_star, ndraws)

    fig, ax = plt.subplots(1, 2, figsize=(16, 5))

    ax[0].hist(tdist, bins=np.max(tdist))
    ax[0].set_title(f"Stopping times over {ndraws} replications")
    ax[0].set(xlabel="time", ylabel="number of stops")
    ax[0].annotate(f"mean = {np.mean(tdist)}", xy=(max(tdist) / 2,
                   max(np.histogram(tdist, bins=max(tdist))[0]) / 2))

    ax[1].hist(cdist.astype(int), bins=2)
    ax[1].set_title(f"Correct decisions over {ndraws} replications")
    ax[1].annotate(f"% correct = {np.mean(cdist)}",
                   xy=(0.05, ndraws / 2))

    plt.show()

simulation_plot(wf)
```

```
  Error at iteration 25 is 8.554240983382755e-05.

Converged in 25 iterations.
```

45.6.3 Comparative Statics

Now let's consider the following exercise.

We double the cost of drawing an additional observation.

Before you look, think about what will happen:

- Will the decision-maker be correct more or less often?
- Will he make decisions sooner or later?

```
In [11]: wf = WaldFriedman(c=2.5)
         simulation_plot(wf)
```

```
Converged in 13 iterations.
```

Increased cost per draw has induced the decision-maker to take less draws before deciding.

Because he decides with less, the percentage of time he is correct drops.

This leads to him having a higher expected loss when he puts equal weight on both models.

45.6.4 A Notebook Implementation

To facilitate comparative statics, we provide a Jupyter notebook that generates the same plots, but with sliders.

With these sliders, you can adjust parameters and immediately observe

- effects on the smoothness of the value function in the indecisive middle range as we increase the number of grid points in the piecewise linear approximation.
- effects of different settings for the cost parameters L_0, L_1, c, the parameters of two beta distributions f_0 and f_1, and the number of points and linear functions m to use in the piece-wise continuous approximation to the value function.
- various simulations from f_0 and associated distributions of waiting times to making a decision.
- associated histograms of correct and incorrect decisions.

45.7 Comparison with Neyman-Pearson Formulation

For several reasons, it is useful to describe the theory underlying the test that Navy Captain G. S. Schuyler had been told to use and that led him to approach Milton Friedman and Allan Wallis to convey his conjecture that superior practical procedures existed.

Evidently, the Navy had told Captail Schuyler to use what it knew to be a state-of-the-art Neyman-Pearson test.

We'll rely on Abraham Wald's [162] elegant summary of Neyman-Pearson theory.

For our purposes, watch for there features of the setup:

- the assumption of a *fixed* sample size n
- the application of laws of large numbers, conditioned on alternative probability models, to interpret the probabilities α and β defined in the Neyman-Pearson theory

Recall that in the sequential analytic formulation above, that

- The sample size n is not fixed but rather an object to be chosen; technically n is a random variable.
- The parameters β and α characterize cut-off rules used to determine n as a random variable.
- Laws of large numbers make no appearances in the sequential construction.

In chapter 1 of **Sequential Analysis** [162] Abraham Wald summarizes the Neyman-Pearson approach to hypothesis testing.

Wald frames the problem as making a decision about a probability distribution that is partially known.

(You have to assume that *something* is already known in order to state a well-posed problem – usually, *something* means *a lot*)

By limiting what is unknown, Wald uses the following simple structure to illustrate the main ideas:

- A decision-maker wants to decide which of two distributions f_0, f_1 govern an IID random variable z.
- The null hypothesis H_0 is the statement that f_0 governs the data.
- The alternative hypothesis H_1 is the statement that f_1 governs the data.
- The problem is to devise and analyze a test of hypothesis H_0 against the alternative hypothesis H_1 on the basis of a sample of a fixed number n independent observations $z_1, z_2, \dots, z_n$ of the random variable z.

To quote Abraham Wald,

> A test procedure leading to the acceptance or rejection of the [null] hypothesis in question is simply a rule specifying, for each possible sample of size n, whether the [null] hypothesis should be accepted or rejected on the basis of the sample. This may also be expressed as follows: A test procedure is simply a subdivision of the totality of all possible samples of size n into two mutually exclusive parts, say part 1 and part 2, together with the application of the rule that the [null] hypothesis be accepted if the observed sample is contained in part 2. Part 1 is also called the critical region. Since part 2 is the totality of all samples of size n which are not included in part 1, part 2 is uniquely determined by part 1. Thus, choosing a test procedure is equivalent to determining a critical region.

Let's listen to Wald longer:

> As a basis for choosing among critical regions the following considerations have been advanced by Neyman and Pearson: In accepting or rejecting H_0 we may commit errors of two kinds. We commit an error of the first kind if we reject H_0 when it is true; we commit an error of the second kind if we accept H_0 when H_1 is true. After a particular critical region W has been chosen, the probability of committing an error of the first kind, as well as the probability of committing an error of the second kind is uniquely determined. The probability of committing an error of the first kind is equal to the probability, determined by the assumption that H_0 is true, that the observed sample will be included in the critical region W. The probability of committing an error of the second kind is equal to the probability, determined on the assumption that H_1 is true, that the probability will fall outside the critical region W. For any given critical region W we shall denote the probability of an error of the first kind by α and the probability of an error of the second kind by β.

Let's listen carefully to how Wald applies law of large numbers to interpret α and β:

> The probabilities α and β have the following important practical interpretation: Suppose that we draw a large number of samples of size n. Let M be the number of such samples drawn. Suppose that for each of these M samples we reject H_0 if the sample is included in W and accept H_0 if the sample lies outside W. In this way we make M statements of rejection or acceptance. Some of these statements will in general be wrong. If H_0 is true and if M is large, the probability is nearly 1 (i.e., it is practically certain) that the proportion of wrong statements (i.e., the number of wrong statements divided by M) will be approximately α. If H_1 is true, the probability is nearly 1 that the proportion of wrong statements will be approximately β. Thus, we can say that in the long run [here Wald applies law of large numbers by driving $M \to \infty$ (our comment, not Wald's)] the proportion of wrong statements will be α if $H_0$is true and β if H_1 is true.

The quantity α is called the *size* of the critical region, and the quantity $1 - \beta$ is called the *power* of the critical region.

Wald notes that

> one critical region W is more desirable than another if it has smaller values of α and β. Although either α or β can be made arbitrarily small by a proper choice of the critical region W, it is possible to make both α and β arbitrarily small for a fixed value of n, i.e., a fixed sample size.

Wald summarizes Neyman and Pearson's setup as follows:

> Neyman and Pearson show that a region consisting of all samples $(z_1, z_2, \dots, z_n)$ which satisfy the inequality

$$\frac{f_1(z_1) \cdots f_1(z_n)}{f_0(z_1) \cdots f_0(z_n)} \geq k$$

is a most powerful critical region for testing the hypothesis H_0 against the alternative hypothesis H_1. The term k on the right side is a constant chosen so that the region will have the required size α.

Wald goes on to discuss Neyman and Pearson's concept of *uniformly most powerful* test.

Here is how Wald introduces the notion of a sequential test

> A rule is given for making one of the following three decisions at any stage of the experiment (at the m th trial for each integral value of m): (1) to accept the hypothesis H , (2) to reject the hypothesis H , (3) to continue the experiment by making an additional observation. Thus, such a test procedure is carried out sequentially. On the basis of the first observation, one of the aforementioned decision is made. If the first or second decision is made, the process is terminated. If the third decision is made, a second trial is performed. Again, on the basis of the first two observations, one of the three decision is made. If the third decision is made, a third trial is performed, and so on. The process is continued until either the first or the second decisions is made. The number n of observations required by such a test procedure is a random variable, since the value of n depends on the outcome of the observations.

Footnotes

[1] The decision maker acts as if he believes that the sequence of random variables $[z_0, z_1, ...]$ is *exchangeable*. See Exchangeability and Bayesian Updating and [100] chapter 11, for discussions of exchangeability.

Chapter 46

Optimal Growth I: The Stochastic Optimal Growth Model

46.1 Contents

- Overview 46.2
- The Model 46.3
- Computation 46.4
- Exercises 46.5
- Solutions 46.6

46.2 Overview

In this lecture, we're going to study a simple optimal growth model with one agent.

The model is a version of the standard one sector infinite horizon growth model studied in

- [149], chapter 2
- [108], section 3.1
- EDTC, chapter 1
- [153], chapter 12

The technique we use to solve the model is dynamic programming.

Our treatment of dynamic programming follows on from earlier treatments in our lectures on shortest paths and job search.

We'll discuss some of the technical details of dynamic programming as we go along.

46.2.1 Code

Regarding code, our implementation in this lecture will focus on clarity and flexibility.

Both of these things are helpful, particularly for those readers still trying to understand the material, but they do cost us some speed — as you will see when you run the code.

In the next lecture we will sacrifice some of this clarity and flexibility in order to accelerate our code with just-in-time (JIT) compilation.

Let's start with some imports:

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from scipy.interpolate import interp1d
        from scipy.optimize import minimize_scalar

        %matplotlib inline
```

46.3 The Model

Consider an agent who owns an amount $y_t \in \mathbb{R}_+ := [0, \infty)$ of a consumption good at time t.

This output can either be consumed or invested.

When the good is invested it is transformed one-for-one into capital.

The resulting capital stock, denoted here by k_{t+1}, will then be used for production.

Production is stochastic, in that it also depends on a shock ξ_{t+1} realized at the end of the current period.

Next period output is

$$y_{t+1} := f(k_{t+1})\xi_{t+1}$$

where $f \colon \mathbb{R}_+ \to \mathbb{R}_+$ is called the production function.

The resource constraint is

$$k_{t+1} + c_t \leq y_t \tag{1}$$

and all variables are required to be nonnegative.

46.3.1 Assumptions and Comments

In what follows,

- The sequence $\{\xi_t\}$ is assumed to be IID.
- The common distribution of each ξ_t will be denoted ϕ.
- The production function f is assumed to be increasing and continuous.
- Depreciation of capital is not made explicit but can be incorporated into the production function.

While many other treatments of the stochastic growth model use k_t as the state variable, we will use y_t.

This will allow us to treat a stochastic model while maintaining only one state variable.

We consider alternative states and timing specifications in some of our other lectures.

46.3.2 Optimization

Taking y_0 as given, the agent wishes to maximize

$$\mathbb{E}\left[\sum_{t=0}^{\infty}\beta^t u(c_t)\right] \tag{2}$$

subject to

$$y_{t+1} = f(y_t - c_t)\xi_{t+1} \quad \text{and} \quad 0 \leq c_t \leq y_t \quad \text{for all } t \tag{3}$$

where

- u is a bounded, continuous and strictly increasing utility function and
- $\beta \in (0,1)$ is a discount factor.

In (3) we are assuming that the resource constraint (1) holds with equality — which is reasonable because u is strictly increasing and no output will be wasted at the optimum.

In summary, the agent's aim is to select a path $c_0, c_1, c_2, \ldots$ for consumption that is

1. nonnegative,
2. feasible in the sense of (1),
3. optimal, in the sense that it maximizes (2) relative to all other feasible consumption sequences, and
4. *adapted*, in the sense that the action c_t depends only on observable outcomes, not on future outcomes such as ξ_{t+1}.

In the present context

- y_t is called the *state* variable — it summarizes the "state of the world" at the start of each period.
- c_t is called the *control* variable — a value chosen by the agent each period after observing the state.

46.3.3 The Policy Function Approach

One way to think about solving this problem is to look for the best **policy function**.

A policy function is a map from past and present observables into current action.

We'll be particularly interested in **Markov policies**, which are maps from the current state y_t into a current action c_t.

For dynamic programming problems such as this one (in fact for any Markov decision process), the optimal policy is always a Markov policy.

In other words, the current state y_t provides a sufficient statistic for the history in terms of making an optimal decision today.

This is quite intuitive, but if you wish you can find proofs in texts such as [149] (section 4.1).

Hereafter we focus on finding the best Markov policy.

In our context, a Markov policy is a function $\sigma \colon \mathbb{R}_+ \to \mathbb{R}_+$, with the understanding that states are mapped to actions via

$$c_t = \sigma(y_t) \quad \text{for all } t$$

In what follows, we will call σ a *feasible consumption policy* if it satisfies

$$0 \leq \sigma(y) \leq y \quad \text{for all} \quad y \in \mathbb{R}_+ \tag{4}$$

In other words, a feasible consumption policy is a Markov policy that respects the resource constraint.

The set of all feasible consumption policies will be denoted by Σ.

Each $\sigma \in \Sigma$ determines a continuous state Markov process $\{y_t\}$ for output via

$$y_{t+1} = f(y_t - \sigma(y_t))\xi_{t+1}, \quad y_0 \text{ given} \tag{5}$$

This is the time path for output when we choose and stick with the policy σ.

We insert this process into the objective function to get

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(c_t)\right] = \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(\sigma(y_t))\right] \tag{6}$$

This is the total expected present value of following policy σ forever, given initial income y_0.

The aim is to select a policy that makes this number as large as possible.

The next section covers these ideas more formally.

46.3.4 Optimality

The σ associated with a given policy σ is the mapping defined by

$$v_\sigma(y) = \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(\sigma(y_t))\right] \tag{7}$$

when $\{y_t\}$ is given by (5) with $y_0 = y$.

In other words, it is the lifetime value of following policy σ starting at initial condition y.

The **value function** is then defined as

$$v^*(y) := \sup_{\sigma \in \Sigma} \; v_\sigma(y) \tag{8}$$

The value function gives the maximal value that can be obtained from state y, after considering all feasible policies.

A policy $\sigma \in \Sigma$ is called **optimal** if it attains the supremum in (8) for all $y \in \mathbb{R}_+$.

46.3.5 The Bellman Equation

With our assumptions on utility and production function, the value function as defined in (8) also satisfies a **Bellman equation**.

For this problem, the Bellman equation takes the form

$$v(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y-c)z)\phi(dz) \right\} \qquad (y \in \mathbb{R}_+) \tag{9}$$

This is a *functional equation in* v.

The term $\int v(f(y-c)z)\phi(dz)$ can be understood as the expected next period value when

- v is used to measure value
- the state is y
- consumption is set to c

As shown in EDTC, theorem 10.1.11 and a range of other texts

> *The value function v^* satisfies the Bellman equation*

In other words, (9) holds when $v = v^*$.

The intuition is that maximal value from a given state can be obtained by optimally trading off

- current reward from a given action, vs
- expected discounted future value of the state resulting from that action

The Bellman equation is important because it gives us more information about the value function.

It also suggests a way of computing the value function, which we discuss below.

46.3.6 Greedy Policies

The primary importance of the value function is that we can use it to compute optimal policies.

The details are as follows.

Given a continuous function v on $\mathbb{R}_+$, we say that $\sigma \in \Sigma$ is v-**greedy** if $\sigma(y)$ is a solution to

$$\max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y-c)z)\phi(dz) \right\} \tag{10}$$

for every $y \in \mathbb{R}_+$.

In other words, $\sigma \in \Sigma$ is v-greedy if it optimally trades off current and future rewards when v is taken to be the value function.

In our setting, we have the following key result

> - A feasible consumption policy is optimal if and only it is v^*-greedy.

The intuition is similar to the intuition for the Bellman equation, which was provided after (9).

See, for example, theorem 10.1.11 of EDTC.

Hence, once we have a good approximation to v^*, we can compute the (approximately) optimal policy by computing the corresponding greedy policy.

The advantage is that we are now solving a much lower dimensional optimization problem.

46.3.7 The Bellman Operator

How, then, should we compute the value function?

One way is to use the so-called **Bellman operator**.

(An operator is a map that sends functions into functions.)

The Bellman operator is denoted by T and defined by

$$Tv(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y-c)z)\phi(dz) \right\} \qquad (y \in \mathbb{R}_+) \tag{11}$$

In other words, T sends the function v into the new function Tv defined by (11).

By construction, the set of solutions to the Bellman equation (9) *exactly coincides with* the set of fixed points of T.

For example, if $Tv = v$, then, for any $y \geq 0$,

$$v(y) = Tv(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y-c)z)\phi(dz) \right\}$$

which says precisely that v is a solution to the Bellman equation.

It follows that v^* is a fixed point of T.

46.3.8 Review of Theoretical Results

One can also show that T is a contraction mapping on the set of continuous bounded functions on $\mathbb{R}_+$ under the supremum distance

$$\rho(g, h) = \sup_{y \geq 0} |g(y) - h(y)|$$

See EDTC, lemma 10.1.18.

Hence, it has exactly one fixed point in this set, which we know is equal to the value function.

It follows that

- The value function v^* is bounded and continuous.
- Starting from any bounded and continuous v, the sequence $v, Tv, T^2v, \ldots$ generated by iteratively applying T converges uniformly to v^*.

This iterative method is called **value function iteration**.

We also know that a feasible policy is optimal if and only if it is v^*-greedy.

It's not too hard to show that a v^*-greedy policy exists (see EDTC, theorem 10.1.11 if you get stuck).

Hence, at least one optimal policy exists.

Our problem now is how to compute it.

46.3.9 Unbounded Utility

The results stated above assume that the utility function is bounded.

In practice economists often work with unbounded utility functions — and so will we.

In the unbounded setting, various optimality theories exist.

Unfortunately, they tend to be case-specific, as opposed to valid for a large range of applications.

Nevertheless, their main conclusions are usually in line with those stated for the bounded case just above (as long as we drop the word "bounded").

Consult, for example, section 12.2 of EDTC, [95] or [114].

46.4 Computation

Let's now look at computing the value function and the optimal policy.

We will use fitted value function iteration, which was described in detail in a previous lecture.

The algorithm will be

1. Begin with an array of values $\{v_1, \dots, v_I\}$ representing the values of some initial function v on the grid points $\{y_1, \dots, y_I\}$.
2. Build a function $\hat{v}$ on the state space $\mathbb{R}_+$ by linear interpolation, based on these data points.
3. Obtain and record the value $T\hat{v}(y_i)$ on each grid point y_i by repeatedly solving (11).
4. Unless some stopping condition is satisfied, set $\{v_1, \dots, v_I\} = \{T\hat{v}(y_1), \dots, T\hat{v}(y_I)\}$ and go to step 2.

46.4.1 Scalar Maximization

To maximize the right hand side of the Bellman equation, we are going to use the `minimize_scalar` routine from SciPy.

Since we are maximizing rather than minmizing, we will use the fact that the maximizer of g on the interval $[a, b]$ is the minimizer of $-g$ on the same interval.

To this end, and to keep the interface tidy, we will wrap `minimize_scalar` in an outer function as follows:

```
In [2]: def maximize(g, a, b, args):
        """
```

```
    Maximize the function g over the interval [a, b].

    We use the fact that the maximizer of g on any interval is
    also the minimizer of -g.  The tuple args collects any extra
    arguments to g.

    Returns the maximal value and the maximizer.
    """

    objective = lambda x: -g(x, *args)
    result = minimize_scalar(objective, bounds=(a, b), method='bounded')
    maximizer, maximum = result.x, -result.fun
    return maximizer, maximum
```

46.4.2 Optimal Growth Model

We will assume for now that ϕ is the distribution of $\exp(\mu + s\zeta)$ when ζ is standard normal.

We will store this and other primitives of the optimal growth model in a class.

The class, defined below, combines both parameters and a method that realizes the right hand side of the Bellman equation (9).

```
In [3]: class OptimalGrowthModel:

            def __init__(self,
                         u,            # utility function
                         f,            # production function
                         β=0.96,       # discount factor
                         μ=0,          # shock location parameter
                         s=0.1,        # shock scale parameter
                         grid_max=4,
                         grid_size=120,
                         shock_size=250,
                         seed=1234):

                self.u, self.f, self.β, self.μ, self.s = u, f, β, μ, s

                # Set up grid
                self.grid = np.linspace(1e-4, grid_max, grid_size)

                # Store shocks (with a seed, so results are reproducible)
                np.random.seed(seed)
                self.shocks = np.exp(μ + s * np.random.randn(shock_size))

            def objective(self, c, y, v_array):
                """
                Right hand side of the Bellman equation.
                """

                u, f, β, shocks = self.u, self.f, self.β, self.shocks

                v = interp1d(self.grid, v_array)

                return u(c) + β * np.mean(v(f(y - c) * shocks))
```

In the second last line we are using linear interpolation.

In the last line, the expectation in (11) is computed via Monte Carlo, using the approximation

$$\int v(f(y-c)z)\phi(dz) \approx \frac{1}{n}\sum_{i=1}^{n} v(f(y-c)\xi_i)$$

where $\{\xi_i\}_{i=1}^n$ are IID draws from ϕ.

Monte Carlo is not always the most efficient way to compute integrals numerically but it does have some theoretical advantages in the present setting.

(For example, it preserves the contraction mapping property of the Bellman operator — see, e.g., [124])

46.4.3 The Bellman Operator

The next function implements the Bellman operator.

(We could have added it as a method to the `OptimalGrowthModel` class, but we prefer small classes rather than monolithic ones for this kind of numerical work.)

```
In [4]: def T(og, v):
            """
            The Bellman operator.  Updates the guess of the value function
            and also computes a v-greedy policy.

              * og is an instance of OptimalGrowthModel
              * v is an array representing a guess of the value function

            """
            v_new = np.empty_like(v)
            v_greedy = np.empty_like(v)

            for i in range(len(grid)):
                y = grid[i]

                # Maximize RHS of Bellman equation at state y
                c_star, v_max = maximize(og.objective, 1e-10, y, (y, v))
                v_new[i] = v_max
                v_greedy[i] = c_star

            return v_greedy, v_new
```

46.4.4 An Example

Let's suppose now that

$$f(k) = k^\alpha \quad \text{and} \quad u(c) = \ln c$$

For this particular problem, an exact analytical solution is available (see [108], section 3.1.2), with

$$v^*(y) = \frac{\ln(1-\alpha\beta)}{1-\beta} + \frac{(\mu + \alpha\ln(\alpha\beta))}{1-\alpha}\left[\frac{1}{1-\beta} - \frac{1}{1-\alpha\beta}\right] + \frac{1}{1-\alpha\beta}\ln y \tag{12}$$

and optimal consumption policy

$$\sigma^*(y) = (1 - \alpha\beta)y$$

It is valuable to have these closed-form solutions because it lets us check whether our code works for this particular case.

In Python, the functions above can be expressed as

In [5]:
```
def v_star(y, α, β, μ):
    """
    True value function
    """
    c1 = np.log(1 - α * β) / (1 - β)
    c2 = (μ + α * np.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
    return c1 + c2 * (c3 - c4) + c4 * np.log(y)

def σ_star(y, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * y
```

Next let's create an instance of the model with the above primitives and assign it to the variable `og`.

In [6]:
```
α = 0.4
def fcd(k):
    return k**α

og = OptimalGrowthModel(u=np.log, f=fcd)
```

Now let's see what happens when we apply our Bellman operator to the exact solution v^* in this case.

In theory, since v^* is a fixed point, the resulting function should again be v^*.

In practice, we expect some small numerical error

In [7]:
```
grid = og.grid

v_init = v_star(grid, α, og.β, og.μ)    # Start at the solution
v_greedy, v = T(og, v_init)             # Apply T once

fig, ax = plt.subplots()
ax.set_ylim(-35, -24)
ax.plot(grid, v, lw=2, alpha=0.6, label='$Tv^*$')
ax.plot(grid, v_init, lw=2, alpha=0.6, label='$v^*$')
ax.legend()
plt.show()
```

The two functions are essentially indistinguishable, so we are off to a good start.

Now let's have a look at iterating with the Bellman operator, starting off from an arbitrary initial condition.

The initial condition we'll start with is, somewhat arbitrarily, $v(y) = 5\ln(y)$

```
In [8]: v = 5 * np.log(grid)  # An initial condition
        n = 35

        fig, ax = plt.subplots()

        ax.plot(grid, v, color=plt.cm.jet(0),
                lw=2, alpha=0.6, label='Initial condition')

        for i in range(n):
            v_greedy, v = T(og, v)  # Apply the Bellman operator
            ax.plot(grid, v, color=plt.cm.jet(i / n), lw=2, alpha=0.6)

        ax.plot(grid, v_star(grid, α, og.β, og.μ), 'k-', lw=2,
                alpha=0.8, label='True value function')

        ax.legend()
        ax.set(ylim=(-40, 10), xlim=(np.min(grid), np.max(grid)))
        plt.show()
```

The figure shows

1. the first 36 functions generated by the fitted value function iteration algorithm, with hotter colors given to higher iterates
2. the true value function v^* drawn in black

The sequence of iterates converges towards v^*.

We are clearly getting closer.

46.4.5 Iterating to Convergence

We can write a function that iterates until the difference is below a particular tolerance level.

```
In [9]: def solve_model(og,
                        tol=1e-4,
                        max_iter=1000,
                        verbose=True,
                        print_skip=25):

            # Set up loop
            v = np.log(og.grid)  # Initial condition
            i = 0
            error = tol + 1

            while i < max_iter and error > tol:
                v_greedy, v_new = T(og, v)
                error = np.max(np.abs(v - v_new))
                i += 1
                if verbose and i % print_skip == 0:
                    print(f"Error at iteration {i} is {error}.")
```

```
            v = v_new

        if i == max_iter:
            print("Failed to converge!")

        if verbose and i < max_iter:
            print(f"\nConverged in {i} iterations.")

        return v_greedy, v_new
```

Let's use this function to compute an approximate solution at the defaults.

```
In [10]: v_greedy, v_solution = solve_model(og)
```

```
    Error at iteration 25 is 0.40975776844490497.
Error at iteration 50 is 0.1476753540823772.
Error at iteration 75 is 0.05322171277213883.
Error at iteration 100 is 0.019180930548646558.
Error at iteration 125 is 0.006912744396029069.
Error at iteration 150 is 0.002491330384817303.
Error at iteration 175 is 0.000897867291303811.
Error at iteration 200 is 0.00032358842396718046.
Error at iteration 225 is 0.00011662020561331587.

Converged in 229 iterations.
```

Now we check our result by plotting it against the true value:

```
In [11]: fig, ax = plt.subplots()

         ax.plot(grid, v_solution, lw=2, alpha=0.6,
                 label='Approximate value function')

         ax.plot(grid, v_star(grid, α, og.β, og.μ), lw=2,
                 alpha=0.6, label='True value function')

         ax.legend()
         ax.set_ylim(-35, -24)
         plt.show()
```

The figure shows that we are pretty much on the money.

46.4.6 The Policy Function

The policy `v_greedy` computed above corresponds to an approximate optimal policy.

The next figure compares it to the exact solution, which, as mentioned above, is $\sigma(y) = (1 - \alpha\beta)y$

```
In [12]: fig, ax = plt.subplots(figsize=(9, 5))

         ax.plot(grid, v_greedy, lw=2,
                 alpha=0.6, label='Approximate policy function')

         ax.plot(grid, σ_star(grid, α, og.β),
                 lw=2, alpha=0.6, label='True policy function')

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

The figure shows that we've done a good job in this instance of approximating the true policy.

46.5 Exercises

46.5.1 Exercise 1

A common choice for utility function in this kind of work is the CRRA specification

$$u(c) = \frac{c^{1-\gamma} - 1}{1 - \gamma}$$

Maintaining the other defaults, including the Cobb-Douglas production function, solve the optimal growth model with this utility specification.

In doing so,

- Set $\gamma = 1.5$.
- Use the `solve_model` function defined above.
- Time how long this function takes to run, so we can compare it to faster code developed in the next lecture

46.6 Solutions

46.6.1 Exercise 1

Here we set up the model.

```
In [13]: γ = 1.5   # Preference parameter
```

```
def u_crra(c):
    return (c**(1 - γ) - 1) / (1 - γ)

og = OptimalGrowthModel(u=u_crra, f=fcd)
```

Now let's run it, with a timer.

```
In [14]: %%time
         v_greedy, v_solution = solve_model(og)
```

```
    Error at iteration 25 is 0.5562115412655544.
Error at iteration 50 is 0.20045587390450237.
Error at iteration 75 is 0.07224363883006646.
Error at iteration 100 is 0.026036370248256446.
Error at iteration 125 is 0.009383422356386006.
Error at iteration 150 is 0.003381754610181531.
Error at iteration 175 is 0.0012187732587420896.
Error at iteration 200 is 0.00043924188108235285.
Error at iteration 225 is 0.00015830133190775086.

Converged in 237 iterations.
CPU times: user 33.3 s, sys: 4.01 ms, total: 33.3 s
Wall time: 33.3 s
```

Let's plot the policy function just to see what it looks like:

```
In [15]: fig, ax = plt.subplots()

         ax.plot(grid, v_greedy, lw=2,
                 alpha=0.6, label='Approximate optimal policy')

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

Chapter 47

Optimal Growth II: Accelerating the Code with Numba

47.1 Contents

- Overview 47.2
- The Model 47.3
- Computation 47.4
- Exercises 47.5
- Solutions 47.6

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

```
In [1]: !pip install --upgrade quantecon
        !pip install --upgrade interpolation
```

47.2 Overview

In a previous lecture, we studied a stochastic optimal growth model with one representative agent.

We solved the model using dynamic programming.

In writing our code, we focused on clarity and flexibility.

These are important, but there's often a trade-off between flexibility and speed.

The reason is that, when code is less flexible, we can exploit structure more easily.

(This is true about algorithms and mathematical problems more generally: more specific problems have more structure, which, with some thought, can be exploited for better results.)

So, in this lecture, we are going to accept less flexibility while gaining speed, using just-in-time (JIT) compilation to accelerate our code.

Let's start with some imports:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        from interpolation import interp
```

```
from numba import jit, njit, jitclass, prange, float64, int32
from quantecon.optimize.scalar_maximization import brent_max

%matplotlib inline
```

We are using an interpolation function from interpolation.py because it helps us JIT-compile our code.

The function brent_max is also designed for embedding in JIT-compiled code.

These are alternatives to similar functions in SciPy (which, unfortunately, are not JIT-aware).

47.3 The Model

The model is the same as discussed in this lecture.

We will use the same algorithm to solve it—the only difference is in the implementation itself.

We will use the CRRA utility specification

$$u(c) = \frac{c^{1-\gamma} - 1}{1 - \gamma}$$

We continue to assume that

- $f(k) = k^\alpha$
- ϕ is the distribution of $\exp(\mu + s\zeta)$ when ζ is standard normal

47.4 Computation

As before, we will store the primitives of the optimal growth model in a class.

But now we are going to use Numba's `@jitclass` decorator to target our class for JIT compilation.

Because we are going to use Numba to compile our class, we need to specify the types of the data:

```
In [3]: opt_growth_data = [
            ('α', float64),          # Production parameter
            ('β', float64),          # Discount factor
            ('μ', float64),          # Shock location parameter
            ('γ', float64),          # Preference parameter
            ('s', float64),          # Shock scale parameter
            ('grid', float64[:]),    # Grid (array)
            ('shocks', float64[:])   # Shock draws (array)
        ]
```

Note the convention for specifying the types of each argument.

Now we're ready to create our class, which will combine parameters and a method that realizes the right hand side of the Bellman equation (9).

You will notice that, unlike in the previous lecture, we hardwire the Cobb-Douglas production and CRRA utility specifications into the class.

Thus, we are losing flexibility, but we will gain substantial speed.

In [4]:
```
@jitclass(opt_growth_data)
class OptimalGrowthModel:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 γ=1.5,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.γ, self.μ, self.s = α, β, γ, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))

    def f(self, k):
        return k**self.α

    def u(self, c):
        return (c**(1 - self.γ) - 1) / (1 - self.γ)

    def objective(self, c, y, v_array):
        """
        Right hand side of the Bellman equation.
        """

        u, f, β, shocks = self.u, self.f, self.β, self.shocks

        v = lambda x: interp(self.grid, v_array, x)

        return u(c) + β * np.mean(v(f(y - c) * shocks))
```

47.4.1 The Bellman Operator

Here's a function that uses JIT compilation to accelerate the Bellman operator

In [5]:
```
@jit(nopython=True)
def T(og, v):
    """
    The Bellman operator.

      * og is an instance of OptimalGrowthModel
      * v is an array representing a guess of the value function
    """
    v_new = np.empty_like(v)
```

```
    for i in range(len(og.grid)):
        y = og.grid[i]

        # Maximize RHS of Bellman equation at state y
        v_max = brent_max(og.objective, 1e-10, y, args=(y, v))[1]
        v_new[i] = v_max

    return v_new
```

Here's another function, very similar to the last, that computes a v-greedy policy:

```
In [6]: @jit(nopython=True)
        def get_greedy(og, v):
            """
            Compute a v-greedy policy.

              * og is an instance of OptimalGrowthModel
              * v is an array representing a guess of the value function
            """
            v_greedy = np.empty_like(v)

            for i in range(len(og.grid)):
                y = og.grid[i]

                # Find maximizer of RHS of Bellman equation at state y
                c_star = brent_max(og.objective, 1e-10, y, args=(y, v))[0]
                v_greedy[i] = c_star

            return v_greedy
```

The last two functions could be merged, as they were in our previous implementation, but we resisted doing so to increase efficiency.

Here's a function that iterates from a starting guess of the value function until the difference between successive iterates is below a particular tolerance level.

```
In [7]: def solve_model(og,
                        tol=1e-4,
                        max_iter=1000,
                        verbose=True,
                        print_skip=25):

            # Set up loop
            v = np.log(og.grid)  # Initial condition
            i = 0
            error = tol + 1

            while i < max_iter and error > tol:
                v_new = T(og, v)
                error = np.max(np.abs(v - v_new))
                i += 1
                if verbose and i % print_skip == 0:
                    print(f"Error at iteration {i} is {error}.")
                v = v_new

            if i == max_iter:
```

```
            print("Failed to converge!")

        if verbose and i < max_iter:
            print(f"\nConverged in {i} iterations.")

        return v_new
```

Let's compute the approximate solution at the default parameters.

First we create an instance:

```
In [8]: og = OptimalGrowthModel()
```

Now we call `solve_model`, using the `%%time` magic to check how long it takes.

```
In [9]: %%time
        v_solution = solve_model(og)
```

```
    Error at iteration 25 is 0.9732383838286296.
Error at iteration 50 is 0.2005788181409116.
Error at iteration 75 is 0.07224367966546197.
Error at iteration 100 is 0.026036371917143697.
Error at iteration 125 is 0.009383422955124843.
Error at iteration 150 is 0.0033817548269325926.
Error at iteration 175 is 0.0012187733382234.
Error at iteration 200 is 0.00043924190907773664.
Error at iteration 225 is 0.00015830134196193058.

Converged in 237 iterations.
CPU times: user 9.07 s, sys: 1.92 ms, total: 9.07 s
Wall time: 9.12 s
```

You will notice that this is *much* faster than our original implementation.

Let's plot the resulting policy:

```
In [10]: v_greedy = get_greedy(og, v_solution)

         fig, ax = plt.subplots()

         ax.plot(og.grid, v_greedy, lw=2,
                 alpha=0.6, label='Approximate value function')

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

Everything seems in order, so our code acceleration has been successful!

47.5 Exercises

47.5.1 Exercise 1

Once an optimal consumption policy σ is given, income follows

$$y_{t+1} = f(y_t - \sigma(y_t))\xi_{t+1}$$

The next figure shows a simulation of 100 elements of this sequence for three different discount factors (and hence three different policies).

In each sequence, the initial condition is $y_0 = 0.1$.

The discount factors are `discount_factors = (0.8, 0.9, 0.98)`.

We have also dialed down the shocks a bit with `s = 0.05`.

Otherwise, the parameters and primitives are the same as the log-linear model discussed earlier in the lecture.

Notice that more patient agents typically have higher wealth.

Replicate the figure modulo randomness.

47.6 Solutions

47.6.1 Exercise 1

Here's one solution

```
In [11]: def simulate_og(σ_func, og, y0=0.1, ts_length=100):
             '''
             Compute a time series given consumption policy σ.
             '''
             y = np.empty(ts_length)
             ξ = np.random.randn(ts_length-1)
             y[0] = y0
             for t in range(ts_length-1):
                 y[t+1] = (y[t] - σ_func(y[t]))**og.α * np.exp(og.μ + og.s * ξ[t])
             return y
```

```
In [12]: fig, ax = plt.subplots()

         for β in (0.8, 0.9, 0.98):
```

```
    og = OptimalGrowthModel(β=β, s=0.05)

    v_solution = solve_model(og)
    v_greedy = get_greedy(og, v_solution)

    # Define an optimal policy function
    σ_func = lambda x: interp(og.grid, v_greedy, x)
    y = simulate_og(σ_func, og)
    ax.plot(y, lw=2, alpha=0.6, label=rf'$\beta = {β}$')

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

```
    Error at iteration 25 is 0.012245923836417205.

Converged in 44 iterations.
Error at iteration 25 is 0.20961181523261985.
Error at iteration 50 is 0.008387575216147525.
Error at iteration 75 is 0.0006017314226482995.

Converged in 93 iterations.
Error at iteration 25 is 1.636332251620388.
Error at iteration 50 is 0.5549102065497209.
Error at iteration 75 is 0.3346444091976082.
Error at iteration 100 is 0.20194598162470356.
Error at iteration 125 is 0.12186727717256929.
Error at iteration 150 is 0.07354260348984099.
Error at iteration 175 is 0.04438036734114803.
Error at iteration 200 is 0.026781986385216783.
Error at iteration 225 is 0.016161984176847.
Error at iteration 250 is 0.009753187414219155.
Error at iteration 275 is 0.005885704607351272.
Error at iteration 300 is 0.003551815140596659.
Error at iteration 325 is 0.0021433951642393367.
Error at iteration 350 is 0.00129346338508185.
Error at iteration 375 is 0.0007805595314920311.
Error at iteration 400 is 0.0004710401467207248.
Error at iteration 425 is 0.0002842561149236644.
Error at iteration 450 is 0.00017153853966078714.
Error at iteration 475 is 0.00010351745852688055.

Converged in 477 iterations.
```

0.6
0.5
0.4
0.3
0.2
0.1
0
20
40
60
80
100
$\beta = 0.8$
$\beta = 0.9$
$\beta = 0.98$

Chapter 48

Optimal Growth III: Time Iteration

48.1 Contents

- Overview 48.2
- The Euler Equation 48.3
- Comparison with Value Function Iteration 48.4
- Implementation 48.5
- Exercises 48.6
- Solutions 48.7

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

```
In [1]: !pip install --upgrade quantecon
        !pip install interpolation
```

48.2 Overview

In this lecture, we'll continue our earlier study of the stochastic optimal growth model.

In that lecture, we solved the associated discounted dynamic programming problem using value function iteration.

The beauty of this technique is its broad applicability.

With numerical problems, however, we can often attain higher efficiency in specific applications by deriving methods that are carefully tailored to the application at hand.

The stochastic optimal growth model has plenty of structure to exploit for this purpose, especially when we adopt some concavity and smoothness assumptions over primitives.

We'll use this structure to obtain an **Euler equation** based method that's more efficient than value function iteration for this and some other closely related applications.

In a subsequent lecture, we'll see that the numerical implementation part of the Euler equation method can be further adjusted to obtain even more efficiency.

Let's start with some imports:

```
In [2]: import numpy as np
        import quantecon as qe
```

```
import matplotlib.pyplot as plt
%matplotlib inline
from quantecon.optimize import brentq
```

48.3 The Euler Equation

Let's take the model set out in the stochastic growth model lecture and add the assumptions that

1. u and f are continuously differentiable and strictly concave
2. $f(0) = 0$
3. $\lim_{c \to 0} u'(c) = \infty$ and $\lim_{c \to \infty} u'(c) = 0$
4. $\lim_{k \to 0} f'(k) = \infty$ and $\lim_{k \to \infty} f'(k) = 0$

The last two conditions are usually called **Inada conditions**.

Recall the Bellman equation

$$v^*(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y-c)z)\phi(dz) \right\} \quad \text{for all} \quad y \in \mathbb{R}_+ \tag{1}$$

Let the optimal consumption policy be denoted by σ^*.

We know that σ^* is a v^* greedy policy so that $\sigma^*(y)$ is the maximizer in (1).

The conditions above imply that

- σ^* is the unique optimal policy for the stochastic optimal growth model
- the optimal policy is continuous, strictly increasing and also **interior**, in the sense that $0 < \sigma^*(y) < y$ for all strictly positive y, and
- the value function is strictly concave and continuously differentiable, with

$$(v^*)'(y) = u'(\sigma^*(y)) := (u' \circ \sigma^*)(y) \tag{2}$$

The last result is called the **envelope condition** due to its relationship with the envelope theorem.

To see why (2) might be valid, write the Bellman equation in the equivalent form

$$v^*(y) = \max_{0 \leq k \leq y} \left\{ u(y-k) + \beta \int v^*(f(k)z)\phi(dz) \right\},$$

differentiate naively with respect to y, and then evaluate at the optimum.

Section 12.1 of EDTC contains full proofs of these results, and closely related discussions can be found in many other texts.

Differentiability of the value function and interiority of the optimal policy imply that optimal consumption satisfies the first order condition associated with (1), which is

$$u'(\sigma^*(y)) = \beta \int (v^*)'(f(y - \sigma^*(y))z) f'(y - \sigma^*(y)) z \phi(dz) \tag{3}$$

Combining (2) and the first-order condition (3) gives the famous **Euler equation**

$$(u' \circ \sigma^*)(y) = \beta \int (u' \circ \sigma^*)(f(y - \sigma^*(y))z) f'(y - \sigma^*(y)) z \phi(dz) \tag{4}$$

We can think of the Euler equation as a functional equation

$$(u' \circ \sigma)(y) = \beta \int (u' \circ \sigma)(f(y - \sigma(y))z) f'(y - \sigma(y)) z \phi(dz) \tag{5}$$

over interior consumption policies σ, one solution of which is the optimal policy σ^*.

Our aim is to solve the functional equation (5) and hence obtain σ^*.

48.3.1 The Coleman-Reffett Operator

Recall the Bellman operator

$$Tw(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y - c)z) \phi(dz) \right\} \tag{6}$$

Just as we introduced the Bellman operator to solve the Bellman equation, we will now introduce an operator over policies to help us solve the Euler equation.

This operator K will act on the set of all $\sigma \in \Sigma$ that are continuous, strictly increasing and interior (i.e., $0 < \sigma(y) < y$ for all strictly positive y).

Henceforth we denote this set of policies by $\mathcal{P}$

1. The operator K takes as its argument a $\sigma \in \mathcal{P}$ and
2. returns a new function $K\sigma$, where $K\sigma(y)$ is the $c \in (0, y)$ that solves.

$$u'(c) = \beta \int (u' \circ \sigma)(f(y - c)z) f'(y - c) z \phi(dz) \tag{7}$$

We call this operator the **Coleman-Reffett operator** to acknowledge the work of [37] and [130].

In essence, $K\sigma$ is the consumption policy that the Euler equation tells you to choose today when your future consumption policy is σ.

The important thing to note about K is that, by construction, its fixed points coincide with solutions to the functional equation (5).

In particular, the optimal policy σ^* is a fixed point.

Indeed, for fixed y, the value $K\sigma^*(y)$ is the c that solves

$$u'(c) = \beta \int (u' \circ \sigma^*)(f(y - c)z) f'(y - c) z \phi(dz)$$

In view of the Euler equation, this is exactly $\sigma^*(y)$.

48.3.2 Is the Coleman-Reffett Operator Well Defined?

In particular, is there always a unique $c \in (0, y)$ that solves (7)?

The answer is yes, under our assumptions.

For any $\sigma \in \mathcal{P}$, the right side of (7)

- is continuous and strictly increasing in c on $(0, y)$
- diverges to $+\infty$ as $c \uparrow y$

The left side of (7)

- is continuous and strictly decreasing in c on $(0, y)$
- diverges to $+\infty$ as $c \downarrow 0$

Sketching these curves and using the information above will convince you that they cross exactly once as c ranges over $(0, y)$.

With a bit more analysis, one can show in addition that $K\sigma \in \mathcal{P}$ whenever $\sigma \in \mathcal{P}$.

48.4 Comparison with Value Function Iteration

How does Euler equation time iteration compare with value function iteration?

Both can be used to compute the optimal policy, but is one faster or more accurate?

There are two parts to this story.

First, on a theoretical level, the two methods are essentially isomorphic.

In particular, they converge at the same rate.

We'll prove this in just a moment.

The other side of the story is the accuracy of the numerical implementation.

It turns out that, once we actually implement these two routines, time iteration is more accurate than value function iteration.

More on this below.

48.4.1 Equivalent Dynamics

Let's talk about the theory first.

To explain the connection between the two algorithms, it helps to understand the notion of equivalent dynamics.

(This concept is very helpful in many other contexts as well)

Suppose that we have a function $g \colon X \to X$ where X is a given set.

The pair (X, g) is sometimes called a **dynamical system** and we associate it with trajectories of the form

$$x_{t+1} = g(x_t), \qquad x_0 \text{ given}$$

Equivalently, $x_t = g^t(x_0)$, where g is the t-th composition of g with itself.

Here's the picture

$$x_0 \xrightarrow{g} g(x_0) \xrightarrow{g} g^2(x_0) \xrightarrow{g} g^3(x_0) \xrightarrow{g} \cdots$$

Now let another function $h\colon Y \to Y$ where Y is another set.

Suppose further that

- there exists a bijection τ from X to Y
- the two functions **commute** under τ, which is to say that $\tau(g(x)) = h(\tau(x))$ for all $x \in X$

The last statement can be written more simply as

$$\tau \circ g = h \circ \tau$$

or, by applying τ^{-1} to both sides

$$g = \tau^{-1} \circ h \circ \tau \tag{8}$$

Here's a commutative diagram that illustrates

$$\begin{tikzcd} X \arrow[r, "g"] \arrow[d, "\tau"'] & X \\ Y \arrow[r, "h"'] & Y \arrow[u, "\tau^{-1}"'] \end{tikzcd}$$

Here's a similar figure that traces out the action of the maps on a point $x \in X$

$$\begin{tikzcd} x \arrow[r, "g"] \arrow[d, "\tau"'] & g(x) \\ \tau(x) \arrow[r, "h"'] & h(\tau(x)) \arrow[u, "\tau^{-1}"'] \end{tikzcd}$$

Now, it's easy to check from (8) that $g^2 = \tau^{-1} \circ h^2 \circ \tau$ holds.

In fact, if you like proofs by induction, you won't have trouble showing that

$$g^n = \tau^{-1} \circ h^n \circ \tau$$

is valid for all n.

What does this tell us?

It tells us that the following are equivalent

- iterate n times with g, starting at x
- shift x to Y using τ, iterate n times with h starting at $\tau(x)$ and shift the result $h^n(\tau(x))$ back to X using τ^{-1}

We end up with exactly the same object.

48.4.2 Back to Economics

Have you guessed where this is leading?

What we're going to show now is that the operators T and K commute under a certain bijection.

The implication is that they have exactly the same rate of convergence.

To make life a little easier, we'll assume in the following analysis (although not always in our applications) that $u(0) = 0$.

A Bijection

Let $\mathcal{V}$ be all strictly concave, continuously differentiable functions v mapping $\mathbb{R}_+$ to itself and satisfying $v(0) = 0$ and $v'(y) > u'(y)$ for all positive y.

For $v \in \mathcal{V}$ let

$$Mv := h \circ v' \qquad \text{where } h := (u')^{-1}$$

Although we omit details, $\sigma := Mv$ is actually the unique v-greedy policy.

- See proposition 12.1.18 of EDTC.

It turns out that M is a bijection from $\mathcal{V}$ to $\mathcal{P}$.

A (solved) exercise below asks you to confirm this.

Commutative Operators

It is an additional solved exercise (see below) to show that T and K commute under M, in the sense that

$$M \circ T = K \circ M \tag{9}$$

In view of the preceding discussion, this implies that

$$T^n = M^{-1} \circ K^n \circ M$$

Hence, T and K converge at exactly the same rate!

48.5 Implementation

We've just shown that the operators T and K have the same rate of convergence.

However, it turns out that, once numerical approximation is taken into account, significant differences arise.

In particular, the image of policy functions under K can be calculated faster and with greater accuracy than the image of value functions under T.

Our intuition for this result is that

- the Coleman-Reffett operator exploits more information because it uses first order and envelope conditions
- policy functions generally have less curvature than value functions, and hence admit more accurate approximations based on grid point information

First, we'll store the parameters of the model in a class `OptimalGrowthModel`

In [3]:
```
class OptimalGrowthModel:

    def __init__(self,
                 f,
                 f_prime,
                 u,
                 u_prime,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 grid_max=4,
                 grid_size=200,
                 shock_size=250):

        self.β, self.μ, self.s = β, μ, s
        self.f, self.u = f, u
        self.f_prime, self.u_prime = f_prime, u_prime

        self.grid = np.linspace(1e-5, grid_max, grid_size)  # Set up grid
        # Store shocks
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))
```

Here's some code that returns the Coleman-Reffett operator, K.

In [4]:
```
import numpy as np
from interpolation import interp
from numba import njit, prange
from quantecon.optimize import brentq


def time_operator_factory(og, parallel_flag=True):
```

```
    """
    A function factory for building the Coleman-Reffett operator.
     Here og is an instance of OptimalGrowthModel.
    """
    β = og.β
    f, u = og.f, og.u
    f_prime, u_prime = og.f_prime, og.u_prime
    grid, shocks = og.grid, og.shocks

    @njit
    def objective(c, σ, y):
        """
        The right hand side of the operator
        """
        # First turn w into a function via interpolation
        σ_func = lambda x: interp(grid, σ, x)
        vals = u_prime(σ_func(f(y - c) * shocks)) * f_prime(y - c) * shocks
        return u_prime(c) - β * np.mean(vals)

    @njit(parallel=parallel_flag)
    def K(σ):
        """
        The Coleman-Reffett operator
        """
        σ_new = np.empty_like(σ)
        for i in prange(len(grid)):
            y = grid[i]
            # Solve for optimal c at y
            c_star = brentq(objective, 1e-10, y-1e-10, args=(σ, y))[0]
            σ_new[i] = c_star

        return σ_new

    return K
```

It has some similarities to the code for the Bellman operator in our optimal growth lecture.

For example, it evaluates integrals by Monte Carlo and approximates functions using linear interpolation.

Here's that Bellman operator code again, which needs to be executed because we'll use it in some tests below.

```
In [5]: import numpy as np
        from interpolation import interp
        from numba import njit, prange
        from quantecon.optimize.scalar_maximization import brent_max


        def operator_factory(og, parallel_flag=True):
            """
            A function factory for building the Bellman operator, as well as
            a function that computes greedy policies.

            Here og is an instance of OptimalGrowthModel.
            """

            f, u, β = og.f, og.u, og.β
```

```
    grid, shocks = og.grid, og.shocks

    @njit
    def objective(c, v, y):
        """
        The right-hand side of the Bellman equation
        """
        # First turn v into a function via interpolation
        v_func = lambda x: interp(grid, v, x)
        return u(c) + β * np.mean(v_func(f(y - c) * shocks))

    @njit(parallel=parallel_flag)
    def T(v):
        """
        The Bellman operator
        """
        v_new = np.empty_like(v)
        for i in prange(len(grid)):
            y = grid[i]
            # Solve for optimal v at y
            v_max = brent_max(objective, 1e-10, y, args=(v, y))[1]
            v_new[i] = v_max
        return v_new

    @njit
    def get_greedy(v):
        """
        Computes the v-greedy policy of a given function v
        """
        σ = np.empty_like(v)
        for i in range(len(grid)):
            y = grid[i]
            # Solve for optimal c at y
            c_max = brent_max(objective, 1e-10, y, args=(v, y))[0]
            σ[i] = c_max
        return σ

    return T, get_greedy
```

48.5.1 Testing on the Log / Cobb–Douglas Case

As we did for value function iteration, let's start by testing our method in the presence of a model that does have an analytical solution.

First, we generate an instance of `OptimalGrowthModel` and return the corresponding Coleman-Reffett operator.

```
In [6]: α = 0.3

        @njit
        def f(k):
            "Deterministic part of production function"
            return k**α

        @njit
        def f_prime(k):
            return α * k**(α - 1)
```

```
og = OptimalGrowthModel(f=f, f_prime=f_prime,
                        u=np.log, u_prime=njit(lambda x: 1/x))

K = time_operator_factory(og)
```

As a preliminary test, let's see if $K\sigma^* = \sigma^*$, as implied by the theory.

In [7]:
```
@njit
def σ_star(y, α, β):
    "True optimal policy"
    return (1 - α * β) * y

grid, β = og.grid, og.β
σ_star_new = K(σ_star(grid, α, β))

fig, ax = plt.subplots()
ax.plot(grid, σ_star(grid, α, β), label="optimal policy $\sigma^*$")
ax.plot(grid, σ_star_new, label="$K\sigma^*$")

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

We can't really distinguish the two plots, so we are looking good, at least for this test.

Next, let's try iterating from an arbitrary initial condition and see if we converge towards σ^*.

The initial condition we'll use is the one that eats the whole pie: $\sigma(y) = y$.

In [8]:
```
n = 15
σ = grid.copy()  # Set initial condition
fig, ax = plt.subplots(figsize=(9, 6))
lb = 'initial condition $\sigma(y) = y$'
```

```
ax.plot(grid, σ, color=plt.cm.jet(0), alpha=0.6, label=lb)

for i in range(n):
    σ = K(σ)
    ax.plot(grid, σ, color=plt.cm.jet(i / n), alpha=0.6)

lb = 'true policy function $\sigma^*$'
ax.plot(grid, σ_star(grid, α, β), 'k-', alpha=0.8, label=lb)
ax.legend()

plt.show()
```

We see that the policy has converged nicely, in only a few steps.

Now let's compare the accuracy of iteration between the operators.

We'll generate

1. $K^n \sigma$ where $\sigma(y) = y$
2. $(M \circ T^n \circ M^{-1})\sigma$ where $\sigma(y) = y$

In each case, we'll compare the resulting policy to σ^*.

The theory on equivalent dynamics says we will get the same policy function and hence the same errors.

But in fact we expect the first method to be more accurate for reasons discussed above

```
In [9]: T, get_greedy = operator_factory(og)  # Return the Bellman operator

        σ = grid          # Set initial condition for σ
```

```
v = og.u(grid)    # Set initial condition for v
sim_length = 20

for i in range(sim_length):
    σ = K(σ)  # Time iteration
    v = T(v)  # Value function iteration

# Calculate difference with actual solution
σ_error = σ_star(grid, α, β) - σ
v_error = σ_star(grid, α, β) - get_greedy(v)

plt.plot(grid, σ_error, alpha=0.6, label="policy iteration error")
plt.plot(grid, v_error, alpha=0.6, label="value iteration error")
plt.legend()
plt.show()
```

As you can see, time iteration is much more accurate for a given number of iterations.

48.6 Exercises

48.6.1 Exercise 1

Show that (9) is valid. In particular,

- Let v be strictly concave and continuously differentiable on $(0, \infty)$.
- Fix $y \in (0, \infty)$ and show that $MTv(y) = KMv(y)$.

48.6.2 Exercise 2

Show that M is a bijection from $\mathcal{V}$ to $\mathcal{P}$.

48.6.3 Exercise 3

Consider the same model as above but with the CRRA utility function

$$u(c) = \frac{c^{1-\gamma} - 1}{1 - \gamma}$$

Iterate 20 times with Bellman iteration and Euler equation time iteration

- start time iteration from $\sigma(y) = y$
- start value function iteration from $v(y) = u(y)$
- set $\gamma = 1.5$

Compare the resulting policies and check that they are close.

48.6.4 Exercise 4

Solve the above model as we did in the previous lecture using the operators T and K, and check the solutions are similiar by plotting.

48.7 Solutions

48.7.1 Exercise 1

Let T, K, M, v and y be as stated in the exercise.

Using the envelope theorem, one can show that $(Tv)'(y) = u'(\sigma(y))$ where $\sigma(y)$ solves

$$u'(\sigma(y)) = \beta \int v'(f(y - \sigma(y))z) f'(y - \sigma(y)) z \phi(dz) \tag{10}$$

Hence $MTv(y) = (u')^{-1}(u'(\sigma(y))) = \sigma(y)$.

On the other hand, $KMv(y)$ is the $\sigma(y)$ that solves

$$\begin{aligned}
u'(\sigma(y)) &= \beta \int (u' \circ (Mv))(f(y - \sigma(y))z) f'(y - \sigma(y)) z \phi(dz) \\
&= \beta \int (u' \circ ((u')^{-1} \circ v'))(f(y - \sigma(y))z) f'(y - \sigma(y)) z \phi(dz) \\
&= \beta \int v'(f(y - \sigma(y))z) f'(y - \sigma(y)) z \phi(dz)
\end{aligned}$$

We see that $\sigma(y)$ is the same in each case.

48.7.2 Exercise 2

We need to show that M is a bijection from $\mathcal{V}$ to $\mathcal{P}$.

To see this, first observe that, in view of our assumptions above, u' is a strictly decreasing continuous bijection from $(0, \infty)$ to itself.

It follows that h has the same properties.

Moreover, for fixed $v \in \mathcal{V}$, the derivative v' is a continuous, strictly decreasing function.

Hence, for fixed $v \in \mathcal{V}$, the map $Mv = h \circ v'$ is strictly increasing and continuous, taking values in $(0, \infty)$.

Moreover, interiority holds because v' strictly dominates u', implying that

$$(Mv)(y) = h(v'(y)) < h(u'(y)) = y$$

In particular, $\sigma(y) := (Mv)(y)$ is an element of $\mathcal{P}$.

To see that each $\sigma \in \mathcal{P}$ has a preimage $v \in \mathcal{V}$ with $Mv = \sigma$, fix any $\sigma \in \mathcal{P}$.

Let $v(y) := \int_0^y u'(\sigma(x))dx$ with $v(0) = 0$.

With a small amount of effort, you will be able to show that $v \in \mathcal{V}$ and $Mv = \sigma$.

It's also true that M is one-to-one on $\mathcal{V}$.

To see this, suppose that v and w are elements of $\mathcal{V}$ satisfying $Mv = Mw$.

Then $v(0) = w(0) = 0$ and $v' = w'$ on $(0, \infty)$.

The fundamental theorem of calculus then implies that $v = w$ on $\mathbb{R}_+$.

48.7.3 Exercise 3

Here's the code, which will execute if you've run all the code above

```
In [10]: γ = 1.5   # Preference parameter

         @njit
         def u(c):
             return (c**(1 - γ) - 1) / (1 - γ)

         @njit
         def u_prime(c):
             return c**(-γ)

         og = OptimalGrowthModel(f=f, f_prime=f_prime, u=u, u_prime=u_prime)

         T, get_greedy = operator_factory(og)
         K = time_operator_factory(og)

         σ = grid        # Initial condition for σ
         v = u(grid)     # Initial condition for v
         sim_length = 20

         for i in range(sim_length):
             σ = K(σ)  # Time iteration
             v = T(v)  # Value function iteration


         plt.plot(grid, σ, alpha=0.6, label="policy iteration")
         plt.plot(grid, get_greedy(v), alpha=0.6, label="value iteration")
         plt.legend()
         plt.show()
```

The policies are indeed close.

48.7.4 Exercise 4

Here's is the function we need to solve the model using value function iteration, copied from the previous lecture

```
In [11]: import numpy as np


         def solve_model(og,
                         use_parallel=True,
                         tol=1e-4,
                         max_iter=1000,
                         verbose=True,
                         print_skip=25):

             T, _ = operator_factory(og, parallel_flag=use_parallel)

             # Set up loop
             v = np.log(og.grid)  # Initial condition
             i = 0
             error = tol + 1

             while i < max_iter and error > tol:
                 v_new = T(v)
                 error = np.max(np.abs(v - v_new))
                 i += 1
                 if verbose and i % print_skip == 0:
                     print(f"Error at iteration {i} is {error}.")
                 v = v_new

             if i == max_iter:
```

```
            print("Failed to converge!")

        if verbose and i < max_iter:
            print(f"\nConverged in {i} iterations.")

        return v_new
```

Similarly, we can write a function that uses K to solve the model.

```
In [12]: def solve_model_time(og,
                              use_parallel=True,
                              tol=1e-4,
                              max_iter=1000,
                              verbose=True,
                              print_skip=25):

             K = time_operator_factory(og, parallel_flag=use_parallel)

             # Set up loop
             σ = og.grid  # Initial condition
             i = 0
             error = tol + 1

             while i < max_iter and error > tol:
                 σ_new = K(σ)
                 error = np.max(np.abs(σ - σ_new))
                 i += 1
                 if verbose and i % print_skip == 0:
                     print(f"Error at iteration {i} is {error}.")
                 σ = σ_new

             if i == max_iter:
                 print("Failed to converge!")

             if verbose and i < max_iter:
                 print(f"\nConverged in {i} iterations.")

             return σ_new
```

Solving both models and plotting

```
In [13]: v_star = solve_model(og)
         σ_star = solve_model_time(og)

         plt.plot(grid, get_greedy(v_star), alpha=0.6, label='Bellman operator')
         plt.plot(grid, σ_star, alpha=0.6, label='Coleman-Reffett operator')
         plt.legend()
         plt.show()
```

```
    Error at iteration 25 is 0.40504752035622005.
Error at iteration 50 is 0.1459777964685145.
Error at iteration 75 is 0.052609918581538295.
Error at iteration 100 is 0.018960441930961025.
Error at iteration 125 is 0.006833281022181836.
Error at iteration 150 is 0.0024626920458103996.
Error at iteration 175 is 0.0008875461280304364.
```

```
Error at iteration 200 is 0.00031986871067601896.
Error at iteration 225 is 0.00011527963316382284.

Converged in 229 iterations.

Converged in 10 iterations.
```

Time iteration is numerically far more accurate for a given number of iterations.

Chapter 49

Optimal Growth IV: The Endogenous Grid Method

49.1 Contents

- Overview 49.2
- Key Idea 49.3
- Implementation 49.4
- Speed 49.5

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

```
In [1]: !pip install --upgrade quantecon
        !pip install interpolation
```

49.2 Overview

We solved the stochastic optimal growth model using

1. value function iteration
2. Euler equation based time iteration

We found time iteration to be significantly more accurate at each step.

In this lecture, we'll look at an ingenious twist on the time iteration technique called the **endogenous grid method** (EGM).

EGM is a numerical method for implementing policy iteration invented by Chris Carroll.

It is a good example of how a clever algorithm can save a massive amount of computer time.

(Massive when we multiply saved CPU cycles on each implementation times the number of implementations worldwide)

The original reference is [32].

Let's start with some standard imports:

In [2]:
```
import numpy as np
import quantecon as qe
from interpolation import interp
from numba import njit, prange
from quantecon.optimize import brentq
import matplotlib.pyplot as plt
%matplotlib inline
```

49.3 Key Idea

Let's start by reminding ourselves of the theory and then see how the numerics fit in.

49.3.1 Theory

Take the model set out in the time iteration lecture, following the same terminology and notation.

The Euler equation is

$$(u' \circ \sigma^*)(y) = \beta \int (u' \circ \sigma^*)(f(y - \sigma^*(y))z) f'(y - \sigma^*(y)) z \phi(dz) \tag{1}$$

As we saw, the Coleman-Reffett operator is a nonlinear operator K engineered so that σ^* is a fixed point of K.

It takes as its argument a continuous strictly increasing consumption policy $\sigma \in \Sigma$.

It returns a new function $K\sigma$, where $(K\sigma)(y)$ is the $c \in (0, \infty)$ that solves

$$u'(c) = \beta \int (u' \circ \sigma)(f(y - c)z) f'(y - c) z \phi(dz) \tag{2}$$

49.3.2 Exogenous Grid

As discussed in the lecture on time iteration, to implement the method on a computer we need a numerical approximation.

In particular, we represent a policy function by a set of values on a finite grid.

The function itself is reconstructed from this representation when necessary, using interpolation or some other method.

Previously, to obtain a finite representation of an updated consumption policy we

- fixed a grid of income points $\{y_i\}$
- calculated the consumption value c_i corresponding to each y_i using (2) and a root-finding routine

Each c_i is then interpreted as the value of the function $K\sigma$ at y_i.

Thus, with the points $\{y_i, c_i\}$ in hand, we can reconstruct $K\sigma$ via approximation.

Iteration then continues…

49.3.3 Endogenous Grid

The method discussed above requires a root-finding routine to find the c_i corresponding to a given income value y_i.

Root-finding is costly because it typically involves a significant number of function evaluations.

As pointed out by Carroll [32], we can avoid this if y_i is chosen endogenously.

The only assumption required is that u' is invertible on $(0, \infty)$.

The idea is this:

First, we fix an *exogenous* grid $\{k_i\}$ for capital $(k = y - c)$.

Then we obtain c_i via

$$c_i = (u')^{-1} \left\{ \beta \int (u' \circ \sigma)(f(k_i)z) \, f'(k_i) \, z \, \phi(dz) \right\} \tag{3}$$

where $(u')^{-1}$ is the inverse function of u'.

Finally, for each c_i we set $y_i = c_i + k_i$.

It is clear that each (y_i, c_i) pair constructed in this manner satisfies (2).

With the points $\{y_i, c_i\}$ in hand, we can reconstruct $K\sigma$ via approximation as before.

The name EGM comes from the fact that the grid $\{y_i\}$ is determined **endogenously**.

49.4 Implementation

Let's implement this version of the Coleman-Reffett operator and see how it performs.

First, we will construct a class `OptimalGrowthModel` to hold the parameters of the model.

```
In [3]: class OptimalGrowthModel:

            """

            The class holds parameters and true value and policy functions.
            """

            def __init__(self,
                         f,                  # Production function
                         f_prime,            # f'(k)
                         u,                  # Utility function
                         u_prime,            # Marginal utility
                         u_prime_inv,        # Inverse marginal utility
                         β=0.96,             # Discount factor
                         μ=0,
                         s=0.1,
                         grid_max=4,
                         grid_size=200,
                         shock_size=250):

                self.β, self.μ, self.s = β, μ, s
```

```
        self.f, self.u = f, u
        self.f_prime, self.u_prime, self.u_prime_inv = f_prime, u_prime, \
            u_prime_inv

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)
        # Store shocks
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))
```

49.4.1 The Operator

Here's an implementation of K using EGM as described above.

Unlike the previous lecture, we do not just-in-time compile the operator because we want to return the policy function.

Despite this, the EGM method is still faster than the standard Coleman-Reffett operator, as we will see later on.

```
In [4]: def egm_operator_factory(og):
            """
            A function factory for building the Coleman-Reffett operator

            Here og is an instance of OptimalGrowthModel.
            """

            f, u, β = og.f, og.u, og.β
            f_prime, u_prime, u_prime_inv = og.f_prime, og.u_prime, og.u_prime_inv
            grid, shocks = og.grid, og.shocks

            def K(σ):
                """
                The Bellman operator

                * σ is a function
                """
               # Allocate memory for value of consumption on endogenous grid points
                c = np.empty_like(grid)

                # Solve for updated consumption value
                for i, k in enumerate(grid):
                    vals = u_prime(σ(f(k) * shocks)) * f_prime(k) * shocks
                    c[i] = u_prime_inv(β * np.mean(vals))

                # Determine endogenous grid
                y = grid + c  # y_i = k_i + c_i

                # Update policy function and return
                σ_new = lambda x: interp(y, c, x)

                return σ_new

            return K
```

Note the lack of any root-finding algorithm.

We'll also run our original implementation, which uses an exogenous grid and requires root-finding, so we can perform some comparisons.

In [5]:
```
import numpy as np
from interpolation import interp
from numba import njit, prange
from quantecon.optimize import brentq


def time_operator_factory(og, parallel_flag=True):
    """
    A function factory for building the Coleman-Reffett operator.
     Here og is an instance of OptimalGrowthModel.
    """
    β = og.β
    f, u = og.f, og.u
    f_prime, u_prime = og.f_prime, og.u_prime
    grid, shocks = og.grid, og.shocks

    @njit
    def objective(c, σ, y):
        """
        The right hand side of the operator
        """
        # First turn w into a function via interpolation
        σ_func = lambda x: interp(grid, σ, x)
        vals = u_prime(σ_func(f(y - c) * shocks)) * f_prime(y - c) * shocks
        return u_prime(c) - β * np.mean(vals)

    @njit(parallel=parallel_flag)
    def K(σ):
        """
        The Coleman-Reffett operator
        """
        σ_new = np.empty_like(σ)
        for i in prange(len(grid)):
            y = grid[i]
            # Solve for optimal c at y
            c_star = brentq(objective, 1e-10, y-1e-10, args=(σ, y))[0]
            σ_new[i] = c_star

        return σ_new

    return K
```

Let's test out the code above on some example parameterizations.

49.4.2 Testing on the Log / Cobb–Douglas Case

As we did for value function iteration and time iteration, let's start by testing our method with the log-linear benchmark.

First, we generate an instance

In [6]:
```
α = 0.4  # Production function parameter
```

```
@njit
def f(k):
    """
    Cobb-Douglas production function
    """
    return k**α

@njit
def f_prime(k):
    """
    First derivative of the production function
    """
    return α * k**(α - 1)

@njit
def u_prime(c):
    return 1 / c

og = OptimalGrowthModel(f=f,
                        f_prime=f_prime,
                        u=np.log,
                        u_prime=u_prime,
                        u_prime_inv=u_prime)
```

Notice that we're passing `u_prime` twice.

The reason is that, in the case of log utility, $u'(c) = (u')^{-1}(c) = 1/c$.

Hence `u_prime` and `u_prime_inv` are the same.

As a preliminary test, let's see if $K\sigma^* = \sigma^*$, as implied by the theory

In [7]:
```
β, grid = og.β, og.grid

def c_star(y):
    "True optimal policy"
    return (1 - α * β) * y

K = egm_operator_factory(og)  # Return the operator K with endogenous grid

fig, ax = plt.subplots(figsize=(9, 6))

ax.plot(grid, c_star(grid), label="optimal policy $\sigma^*$")
ax.plot(grid, K(c_star)(grid), label="$K\sigma^*$")

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

We can't really distinguish the two plots.

In fact it's easy to see that the difference is essentially zero:

```
In [8]: max(abs(K(c_star)(grid) - c_star(grid)))
```

```
Out[8]: 9.881666666666669e-06
```

Next, let's try iterating from an arbitrary initial condition and see if we converge towards σ^*.

Let's start from the consumption policy that eats the whole pie: $\sigma(y) = y$

```
In [9]: σ = lambda x: x
        n = 15
        fig, ax = plt.subplots(figsize=(9, 6))

        ax.plot(grid, σ(grid), color=plt.cm.jet(0),
                alpha=0.6, label='initial condition $\sigma(y) = y$')

        for i in range(n):
            σ = K(σ)  # Update policy
            ax.plot(grid, σ(grid), color=plt.cm.jet(i / n),  alpha=0.6)

        ax.plot(grid, c_star(grid), 'k-',
                alpha=0.8, label='true policy function $\sigma^*$')

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

We see that the policy has converged nicely, in only a few steps.

49.5 Speed

Now let's compare the clock times per iteration for the standard Coleman-Reffett operator (with exogenous grid) and the EGM version.

We'll do so using the CRRA model adopted in the exercises of the Euler equation time iteration lecture.

In [10]:
```
γ = 1.5   # Preference parameter

@njit
def u(c):
    return (c**(1 - γ) - 1) / (1 - γ)

@njit
def u_prime(c):
    return c**(-γ)

@njit
def u_prime_inv(c):
    return c**(-1 / γ)

og = OptimalGrowthModel(f=f,
                        f_prime=f_prime,
                        u=u,
                        u_prime=u_prime,
                        u_prime_inv=u_prime_inv)

# Standard Coleman-Reffett operator
```

```
K_time = time_operator_factory(og)
# Call once to compile jitted version
K_time(grid)
# Coleman-Reffett operator with endogenous grid
K_egm = egm_operator_factory(og)
```

Here's the result

In [11]:
```
sim_length = 20

print("Timing standard Coleman policy function iteration")
σ = grid    # Initial policy
qe.util.tic()
for i in range(sim_length):
    σ_new = K_time(σ)
    σ = σ_new
qe.util.toc()

print("Timing policy function iteration with endogenous grid")
σ = lambda x: x  # Initial policy
qe.util.tic()
for i in range(sim_length):
    σ_new = K_egm(σ)
    σ = σ_new
qe.util.toc()
```

```
Timing standard Coleman policy function iteration
TOC: Elapsed: 0:00:1.94
Timing policy function iteration with endogenous grid
TOC: Elapsed: 0:00:0.34
```

Out[11]:
```
0.34094905853271484
```

We see that the EGM version is significantly faster, even without jit compilation!

The absence of numerical root-finding means that it is typically more accurate at each step as well.

Chapter 50

The Income Fluctuation Problem

50.1 Contents

- Overview 50.2
- The Optimal Savings Problem 50.3
- Computation 50.4
- Exercises 50.5
- Solutions 50.6

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

```
In [1]: !pip install --upgrade quantecon
        !pip install interpolation
```

50.2 Overview

Next, we study an optimal savings problem for an infinitely lived consumer—the "common ancestor" described in [108], section 1.3.

This is an essential sub-problem for many representative macroeconomic models

- [6]
- [88]
- etc.

It is related to the decision problem in the stochastic optimal growth model and yet differs in important ways.

For example, the choice problem for the agent includes an additive income term that leads to an occasionally binding constraint.

Our presentation of the model will be relatively brief.

- For further details on economic intuition, implication and models, see [108].
- Proofs of all mathematical results stated below can be found in this paper.

To solve the model we will use Euler equation based time iteration, similar to this lecture.

This method turns out to be globally convergent under mild assumptions, even when utility is unbounded (both above and below).

We'll need the following imports:

In [2]:
```
import numpy as np
from quantecon.optimize import brent_max, brentq
from interpolation import interp
from numba import njit
import matplotlib.pyplot as plt
%matplotlib inline
from quantecon import MarkovChain
```

50.2.1 References

Other useful references include [41], [43], [101], [127], [131] and [143].

50.3 The Optimal Savings Problem

Let's write down the model and then discuss how to solve it.

50.3.1 Set-Up

Consider a household that chooses a state-contingent consumption plan $\{c_t\}_{t \geq 0}$ to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$c_t + a_{t+1} \leq R a_t + z_t, \qquad c_t \geq 0, \qquad a_t \geq -b \qquad t = 0, 1, \ldots \tag{1}$$

Here

- $\beta \in (0, 1)$ is the discount factor
- a_t is asset holdings at time t, with ad-hoc borrowing constraint $a_t \geq -b$
- c_t is consumption
- z_t is non-capital income (wages, unemployment compensation, etc.)
- $R := 1 + r$, where $r > 0$ is the interest rate on savings

Non-capital income $\{z_t\}$ is assumed to be a Markov process taking values in $Z \subset (0, \infty)$ with stochastic kernel Π.

This means that $\Pi(z, B)$ is the probability that $z_{t+1} \in B$ given $z_t = z$.

The expectation of $f(z_{t+1})$ given $z_t = z$ is written as

$$\int f(\acute{z}) \, \Pi(z, d\acute{z})$$

We further assume that

1. $r > 0$ and $\beta R < 1$

2. u is smooth, strictly increasing and strictly concave with $\lim_{c\to 0} u'(c) = \infty$ and $\lim_{c\to\infty} u'(c) = 0$

The asset space is $[-b, \infty)$ and the state is the pair $(a, z) \in S := [-b, \infty) \times Z$.

A *feasible consumption path* from $(a, z) \in S$ is a consumption sequence $\{c_t\}$ such that $\{c_t\}$ and its induced asset path $\{a_t\}$ satisfy

1. $(a_0, z_0) = (a, z)$
2. the feasibility constraints in (1), and
3. measurability of c_t w.r.t. the filtration generated by $\{z_1, \dots, z_t\}$

The meaning of the third point is just that consumption at time t can only be a function of outcomes that have already been observed.

50.3.2 Value Function and Euler Equation

The *value function* $V \colon S \to \mathbb{R}$ is defined by

$$V(a, z) := \sup \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\} \tag{2}$$

where the supremum is overall feasible consumption paths from (a, z).

An *optimal consumption path* from (a, z) is a feasible consumption path from (a, z) that attains the supremum in (2).

To pin down such paths we can use a version of the Euler equation, which in the present setting is

$$u'(c_t) \geq \beta R \, \mathbb{E}_t[u'(c_{t+1})] \tag{3}$$

and

$$u'(c_t) = \beta R \, \mathbb{E}_t[u'(c_{t+1})] \quad \text{whenever } c_t < Ra_t + z_t + b \tag{4}$$

In essence, this says that the natural "arbitrage" relation $u'(c_t) = \beta R \, \mathbb{E}_t[u'(c_{t+1})]$ holds when the choice of current consumption is interior.

Interiority means that c_t is strictly less than its upper bound $Ra_t + z_t + b$.

(The lower boundary case $c_t = 0$ never arises at the optimum because $u'(0) = \infty$)

When c_t does hit the upper bound $Ra_t + z_t + b$, the strict inequality $u'(c_t) > \beta R \, \mathbb{E}_t[u'(c_{t+1})]$ can occur because c_t cannot increase sufficiently to attain equality.

With some thought and effort, one can show that (3) and (4) are equivalent to

$$u'(c_t) = \max \left\{ \beta R \, \mathbb{E}_t[u'(c_{t+1})] \, , \; u'(Ra_t + z_t + b) \right\} \tag{5}$$

50.3.3 Optimality Results

Given our assumptions, it is known that

1. For each $(a, z) \in S$, a unique optimal consumption path from (a, z) exists
2. This path is the unique feasible path from (a, z) satisfying the Euler equality (5) and the transversality condition

$$\lim_{t \to \infty} \beta^t \, \mathbb{E} \left[u'(c_t) a_{t+1} \right] = 0 \tag{6}$$

Moreover, there exists an *optimal consumption function* $\sigma^* \colon S \to [0, \infty)$ such that the path from (a, z) generated by

$$(a_0, z_0) = (a, z), \quad z_{t+1} \sim \Pi(z_t, dy), \quad c_t = \sigma^*(a_t, z_t) \quad \text{and} \quad a_{t+1} = R a_t + z_t - c_t$$

satisfies both (5) and (6), and hence is the unique optimal path from (a, z).

In summary, to solve the optimization problem, we need to compute σ^*.

50.4 Computation

There are two standard ways to solve for σ^*

1. Time iteration (TI) using the Euler equality
2. Value function iteration (VFI)

Let's look at these in turn.

50.4.1 Time Iteration

We can rewrite (5) to make it a statement about functions rather than random variables.

In particular, consider the functional equation

$$u' \circ \sigma \, (a, z) = \max \left\{ \gamma \int u' \circ \sigma \left\{ R a + z - c(a, z), \, \acute{z} \right\} \Pi(z, d\acute{z}) \, , \; u'(R a + z + b) \right\} \tag{7}$$

where $\gamma := \beta R$ and $u' \circ c(s) := u'(c(s))$.

Equation (7) is a functional equation in σ.

In order to identify a solution, let $\mathcal{C}$ be the set of candidate consumption functions $\sigma \colon S \to \mathbb{R}$ such that

- each $\sigma \in \mathcal{C}$ is continuous and (weakly) increasing
- $\min Z \leq c(a, z) \leq R a + z + b$ for all $(a, z) \in S$

In addition, let $K \colon \mathcal{C} \to \mathcal{C}$ be defined as follows.

For given $\sigma \in \mathcal{C}$, the value $K\sigma(a, z)$ is the unique $t \in J(a, z)$ that solves

$$u'(t) = \max \left\{ \gamma \int u' \circ \sigma \left\{ Ra + z - t,\, \acute{z} \right\} \Pi(z, d\acute{z}) \,,\; u'(Ra + z + b) \right\} \tag{8}$$

where

$$J(a, z) := \{ t \in \mathbb{R} \,:\, \min Z \leq t \leq Ra + z + b \} \tag{9}$$

We refer to K as Coleman's policy function operator [37].

It is known that

- K is a contraction mapping on $\mathcal{C}$ under the metric

$$\rho(c, d) := \| u' \circ \sigma_1 - u' \circ \sigma_2 \| := \sup_{s \in S} | u'(\sigma_1(s)) - u'(\sigma_2(s)) | \qquad (\sigma_1, \sigma_2 \in \mathcal{C})$$

- The metric ρ is complete on $\mathcal{C}$
- Convergence in ρ implies uniform convergence on compacts

In consequence, K has a unique fixed point $\sigma^* \in \mathcal{C}$ and $K^n c \to \sigma^*$ as $n \to \infty$ for any $\sigma \in \mathcal{C}$.

By the definition of K, the fixed points of K in $\mathcal{C}$ coincide with the solutions to (7) in $\mathcal{C}$.

In particular, it can be shown that the path $\{c_t\}$ generated from $(a_0, z_0) \in S$ using policy function σ^* is the unique optimal path from $(a_0, z_0) \in S$.

TL;DR The unique optimal policy can be computed by picking any $\sigma \in \mathcal{C}$ and iterating with the operator K defined in (8).

50.4.2 Value Function Iteration

The Bellman operator for this problem is given by

$$Tv(a, z) = \max_{0 \leq \sigma \leq Ra + z + b} \left\{ u(c) + \beta \int v(Ra + z - \sigma, \acute{z}) \Pi(z, d\acute{z}) \right\} \tag{10}$$

We have to be careful with VFI (i.e., iterating with T) in this setting because u is not assumed to be bounded

- In fact typically unbounded both above and below — e.g. $u(c) = \log c$.
- In which case, the standard DP theory does not apply.
- $T^n v$ is not guaranteed to converge to the value function for arbitrary continuous bounded v.

Nonetheless, we can always try the popular strategy "iterate and hope".

We can then check the outcome by comparing with that produced by TI.

The latter is known to converge, as described above.

50.4.3 Implementation

First, we build a class called `ConsumerProblem` that stores the model primitives.

In [3]:
```
class ConsumerProblem:
    """
    A class that stores primitives for the income fluctuation problem. The
    income process is assumed to be a finite state Markov chain.
    """
    def __init__(self,
                 r=0.01,                          # Interest rate
                 β=0.96,                          # Discount factor
                 Π=((0.6, 0.4),
                    (0.05, 0.95)),                # Markov matrix for z_t
                 z_vals=(0.5, 1.0),               # State space of z_t
                 b=0,                             # Borrowing constraint
                 grid_max=16,
                 grid_size=50,
                 u=np.log,                        # Utility function
                 du=njit(lambda x: 1/x)):         # Derivative of utility

        self.u, self.du = u, du
        self.r, self.R = r, 1 + r
        self.β, self.b = β, b
        self.Π, self.z_vals = np.array(Π), tuple(z_vals)
        self.asset_grid = np.linspace(-b, grid_max, grid_size)
```

The function `operator_factory` returns the operator `K` as specified above

In [4]:
```
def operator_factory(cp):
    """
    A function factory for building operator K.

    Here cp is an instance of ConsumerProblem.
    """
    # Simplify names, set up arrays
    R, Π, β, u, b, du = cp.R, cp.Π, cp.β, cp.u, cp.b, cp.du
    asset_grid, z_vals = cp.asset_grid, cp.z_vals
    γ = R * β


    @njit
    def euler_diff(c, a, z, i_z, σ):
        """
        The difference of the left-hand side and the right-hand side
        of the Euler Equation.
        """
        lhs = du(c)
        expectation = 0
        for i in range(len(z_vals)):
            expectation += du(interp(asset_grid, σ[:, i], R * a + z - c)) \
                * Π[i_z, i]
        rhs = max(γ * expectation, du(R * a + z + b))

        return lhs - rhs

    @njit
    def K(σ):
        """
        The operator K.
```

```
        Iteration with this operator corresponds to time iteration on the
       Euler equation.  Computes and returns the updated consumption policy
        σ.  The array σ is replaced with a function cf that implements
        univariate linear interpolation over the asset grid for each
        possible value of z.
        """
        σ_new = np.empty_like(σ)
        for i_a in range(len(asset_grid)):
            a = asset_grid[i_a]
            for i_z in range(len(z_vals)):
                z = z_vals[i_z]
                c_star = brentq(euler_diff, 1e-8, R * a + z + b, \
                    args=(a, z, i_z, σ)).root
                σ_new[i_a, i_z] = c_star

        return σ_new

    return K
```

K uses linear interpolation along the asset grid to approximate the value and consumption functions.

To solve for the optimal policy function, we will write a function `solve_model` to iterate and find the optimal σ.

```
In [5]: def solve_model(cp,
                        tol=1e-4,
                        max_iter=1000,
                        verbose=True,
                        print_skip=25):

    """
    Solves for the optimal policy using time iteration

    * cp is an instance of ConsumerProblem
    """

    u, β, b, R = cp.u, cp.β, cp.b, cp.R
    asset_grid, z_vals = cp.asset_grid, cp.z_vals

    # Initial guess of σ
    σ = np.empty((len(asset_grid), len(z_vals)))
    for i_a, a in enumerate(asset_grid):
        for i_z, z in enumerate(z_vals):
            c_max = R * a + z + b
            σ[i_a, i_z] = c_max

    K = operator_factory(cp)

    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        σ_new = K(σ)
        error = np.max(np.abs(σ - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
```

```
            print(f"Error at iteration {i} is {error}.")
        σ = σ_new

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

    return σ_new
```

Plotting the result using the default parameters of the `ConsumerProblem` class

```
In [6]: cp = ConsumerProblem()
        σ_star = solve_model(cp)

        fig, ax = plt.subplots(figsize=(10, 6))
        ax.plot(cp.asset_grid, σ_star[:, 0], label='$\sigma^*$')
        ax.set(xlabel='asset level', ylabel='optimal consumption')
        ax.legend()
        plt.show()
```

```
Error at iteration 25 is 0.007773142982545167.

Converged in 41 iterations.
```

The following exercises walk you through several applications where policy functions are computed.

50.5 Exercises

50.5.1 Exercise 1

Next, let's consider how the interest rate affects consumption.

Reproduce the following figure, which shows (approximately) optimal consumption policies for different interest rates

- Other than `r`, all parameters are at their default values.
- `r` steps through `np.linspace(0, 0.04, 4)`.
- Consumption is plotted against assets for income shock fixed at the smallest value.

The figure shows that higher interest rates boost savings and hence suppress consumption.

50.5.2 Exercise 2

Now let's consider the long run asset levels held by households.

We'll take `r = 0.03` and otherwise use default parameters.

The following figure is a 45 degree diagram showing the law of motion for assets when consumption is optimal

```
In [7]: m = ConsumerProblem(r=0.03, grid_max=4)
        K = operator_factory(m)

        σ_star = solve_model(m, verbose=False)
        a = m.asset_grid
        R, z_vals = m.R, m.z_vals
```

```
fig, ax = plt.subplots(figsize=(10, 8))
ax.plot(a, R * a + z_vals[0] - σ_star[:, 0], label='Low income')
ax.plot(a, R * a + z_vals[1] - σ_star[:, 1], label='High income')
ax.plot(a, a, 'k--')
ax.set(xlabel='Current assets',
       ylabel='Next period assets',
       xlim=(0, 4), ylim=(0, 4))
ax.legend()
plt.show()
```

The blue line and orange line represent the function

$$a' = h(a, z) := Ra + z - \sigma^*(a, z)$$

when income z takes its high and low values respectively.

The dashed line is the 45 degree line.

We can see from the figure that the dynamics will be stable — assets do not diverge.

In fact there is a unique stationary distribution of assets that we can calculate by simulation

- Can be proved via theorem 2 of [86].
- Represents the long run dispersion of assets across households when households have idiosyncratic shocks.

Ergodicity is valid here, so stationary probabilities can be calculated by averaging over a single long time series.

Hence to approximate the stationary distribution we can simulate a long time series for assets and histogram, as in the following figure

Your task is to replicate the figure

- Parameters are as discussed above.
- The histogram in the figure used a single time series $\{a_t\}$ of length 500,000.
- Given the length of this time series, the initial condition (a_0, z_0) will not matter.
- You might find it helpful to use the `MarkovChain` class from `quantecon`.

50.5.3 Exercise 3

Following on from exercises 1 and 2, let's look at how savings and aggregate asset holdings vary with the interest rate

- Note: [108] section 18.6 can be consulted for more background on the topic treated in this exercise.

For a given parameterization of the model, the mean of the stationary distribution can be interpreted as aggregate capital in an economy with a unit mass of *ex-ante* identical households facing idiosyncratic shocks.

Let's look at how this measure of aggregate capital varies with the interest rate and borrowing constraint.

The next figure plots aggregate capital against the interest rate for `b in (1, 3)`

As is traditional, the price (interest rate) is on the vertical axis.

The horizontal axis is aggregate capital computed as the mean of the stationary distribution.

Exercise 3 is to replicate the figure, making use of code from previous exercises.

Try to explain why the measure of aggregate capital is equal to $-b$ when $r = 0$ for both cases shown here.

50.6 Solutions

50.6.1 Exercise 1

```
In [8]: r_vals = np.linspace(0, 0.04, 4)

        fig, ax = plt.subplots(figsize=(10, 8))
        for r_val in r_vals:
            cp = ConsumerProblem(r=r_val)
            σ_star = solve_model(cp, verbose=False)
            ax.plot(cp.asset_grid, σ_star[:, 0], label=f'$r = {r_val:.3f}$')

        ax.set(xlabel='asset level', ylabel='consumption (low income)')
        ax.legend()
        plt.show()
```

50.6.2 Exercise 2

```
In [9]: def compute_asset_series(cp, T=500000, verbose=False):
            """
            Simulates a time series of length T for assets, given optimal
            savings behavior.

            cp is an instance of ConsumerProblem
            """
            Π, z_vals, R = cp.Π, cp.z_vals, cp.R  # Simplify names
            mc = MarkovChain(Π)
            σ_star = solve_model(cp, verbose=False)
            cf = lambda a, i_z: interp(cp.asset_grid, σ_star[:, i_z], a)
            a = np.zeros(T+1)
            z_seq = mc.simulate(T)
            for t in range(T):
                i_z = z_seq[t]
                a[t+1] = R * a[t] + z_vals[i_z] - cf(a[t], i_z)
            return a

        cp = ConsumerProblem(r=0.03, grid_max=4)
        a = compute_asset_series(cp)

        fig, ax = plt.subplots(figsize=(10, 8))
        ax.hist(a, bins=20, alpha=0.5, density=True)
        ax.set(xlabel='assets', xlim=(-0.05, 0.75))
        plt.show()
```

50.6.3 Exercise 3

In [10]:
```
M = 25
r_vals = np.linspace(0, 0.04, M)
fig, ax = plt.subplots(figsize=(10, 8))

for b in (1, 3):
    asset_mean = []
    for r_val in r_vals:
        cp = ConsumerProblem(r=r_val, b=b)
        mean = np.mean(compute_asset_series(cp, T=250000))
        asset_mean.append(mean)
    ax.plot(asset_mean, r_vals, label=f'$b = {b:d}$')
    print(f"Finished iteration b = {b:d}")

ax.set(xlabel='capital', ylabel='interest rate')
ax.grid()
ax.legend()
plt.show()
```

```
Finished iteration b = 1
Finished iteration b = 3
```

$b = 1$
$b = 3$
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
interest rate
−3
−2
−1
0
1
capital

Chapter 51

Discrete State Dynamic Programming

51.1 Contents

- Overview 51.2
- Discrete DPs 51.3
- Solving Discrete DPs 51.4
- Example: A Growth Model 51.5
- Exercises 51.6
- Solutions 51.7
- Appendix: Algorithms 51.8

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

```
In [1]: !pip install --upgrade quantecon
```

51.2 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:

In [2]:
```
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
```

51.2.1 How to Read this Lecture

We use dynamic programming many applied lectures, such as

- The shortest path lecture
- The McCall search model lecture
- The optimal growth 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.

51.2.2 Code

The code discussed below was authored primarily by Daisuke Oyama.

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.

51.2.3 References

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

- [108]
- [84], section 3.5

- [126]
- [149]
- [136]
- [118]
- EDTC, chapter 5

51.3 Discrete DPs

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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t r(s_t, a_t) \tag{1}$$

where

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

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

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

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

Examples:

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

51.3.1 Policies

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

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

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

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

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

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

- See section 5.5 of [126] 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 (1).

Let's now step through these ideas more carefully.

51.3.2 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*.

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

1. A *reward function* $r \colon SA \to \mathbb{R}$.
2. A *transition probability function* $Q \colon SA \to \Delta(S)$, where $\Delta(S)$ is the set of probability distributions over S.
3. 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 Σ.

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

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

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

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

Notice that Q_σ is a stochastic matrix on S.

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

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

$$(Q_\sigma^t r_\sigma)(s) = \mathbb{E}[r(s_t, \sigma(s_t)) \mid s_0 = s] \quad \text{when } \{s_t\} \sim Q_\sigma \tag{2}$$

Comments

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

51.3.3 Value and Optimality

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

To calculate this quantity we pass the expectation through the sum in (1) and use (2) to get

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

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

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

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

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

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

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

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

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

51.3.4 Two Operators

It is useful to define the following operators:

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

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

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

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

This can be written more succinctly in operator notation as

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

The two operators are both monotone

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

They are also contraction mappings with modulus β

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

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

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

51.3.5 The Bellman Equation and the Principle of Optimality

The main principle of the theory of dynamic programming is that

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

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

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

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

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

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

51.4 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.

51.4.1 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.

51.4.2 Policy Function Iteration

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

Each iteration consists of

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

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

- See theorem 10.2.6 of EDTC for a proof.

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

51.4.3 Modified Policy Function Iteration

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

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

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

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

51.5 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 $\beta \in [0, 1)$.

51.5.1 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

$$Q(s, a, s') := \begin{cases} \frac{1}{B+1} & \text{if } a \leq s' \leq a + B \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

51.5.2 Defining a DiscreteDP Instance

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

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

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

For Q we follow the rule in (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

In [3]:
```
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`.

```
In [4]: 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`

```
In [5]: 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

```
In [6]: results = ddp.solve(method='policy_iteration')
```

Let's see what we've got here

```
In [7]: dir(results)

Out[7]: ['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

```
In [8]: results.v

Out[8]: 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])

In [9]: results.sigma
```

```
Out[9]: 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

```
In [10]: results.max_iter
```

```
Out[10]: 250
```

```
In [11]: results.num_iter
```

```
Out[11]: 3
```

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

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

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

```
In [12]: results.mc.stationary_distributions
```

```
Out[12]: 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?

```
In [13]: 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
```

```
Out[13]: 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

51.5.3 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

```
In [14]: B, M, α, β = 10, 5, 0.5, 0.9
         n = B + M + 1
         m = M + 1

         def u(c):
             return c**α

         s_indices = []
         a_indices = []
         Q = []
         R = []
```

```
b = 1.0 / (B + 1)

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

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

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

51.6 Exercises

In the stochastic optimal growth lecture dynamic programming lecture, we solve a benchmark model that has an analytical solution to check we could replicate it numerically.

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

51.7 Solutions

Written jointly with Diasuke Oyama.

51.7.1 Setup

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

As in the lecture, we let $f(k) = k^\alpha$ with $\alpha = 0.65$, $u(c) = \log c$, and $\beta = 0.95$

In [15]:
```
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`

In [16]:
```
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:

```
In [17]: # Consumption matrix, with nonpositive consumption included
         C = f(grid).reshape(grid_size, 1) - grid.reshape(1, grid_size)

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

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

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

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

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

```
In [18]: 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):

```
In [19]: 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)

```
In [20]: # data = np.ones(L)
         # indptr = np.arange(L+1)
         # Q = sparse.csr_matrix((data, a_indices, indptr), shape=(L, grid_size))
```

Discrete growth model:

```
In [21]: 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.

51.7.2 Solving the Model

Solve the dynamic optimization problem:

```
In [22]: res = ddp.solve(method='policy_iteration')
         v, σ, num_iter = res.v, res.sigma, res.num_iter
         num_iter
```

```
Out[22]: 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.

```
In [23]: # Optimal consumption in the discrete version
         c = f(grid) - grid[σ]

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

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

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

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

```
In [24]: 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:

```
In [25]: np.abs(v - v_star(grid)).max()

Out[25]: 121.49819147053378

In [26]: np.abs(v - v_star(grid))[1:].max()

Out[26]: 0.012681735127500815
```

The optimal consumption functions are close as well:

```
In [27]: np.abs(c - c_star(grid)).max()

Out[27]: 0.003826523100010082
```

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

```
In [28]: diff = np.diff(c)
         (diff >= 0).all()

Out[28]: False

In [29]: dec_ind = np.where(diff < 0)[0]
         len(dec_ind)

Out[29]: 174

In [30]: np.abs(diff[dec_ind]).max()

Out[30]: 0.001961853339766839
```

The value function is monotone:

```
In [31]: (np.diff(v) > 0).all()

Out[31]: True
```

51.7.3 Comparison of the Solution Methods

Let us solve the problem with the other two methods.

Value Iteration

```
In [32]: ddp.epsilon = 1e-4
         ddp.max_iter = 500
         res1 = ddp.solve(method='value_iteration')
         res1.num_iter
```

```
Out[32]: 294
```

```
In [33]: np.array_equal(σ, res1.sigma)
```

```
Out[33]: True
```

Modified Policy Iteration

```
In [34]: res2 = ddp.solve(method='modified_policy_iteration')
         res2.num_iter
```

```
Out[34]: 16
```

```
In [35]: np.array_equal(σ, res2.sigma)
```

```
Out[35]: True
```

Speed Comparison

```
In [36]: %timeit ddp.solve(method='value_iteration')
         %timeit ddp.solve(method='policy_iteration')
         %timeit ddp.solve(method='modified_policy_iteration')
```

```
    286 ms ± 4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
25.5 ms ± 1.49 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
29.2 ms ± 1.05 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

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

51.7.4 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`

```
In [37]: w = 5 * np.log(grid) - 25  # Initial condition
         n = 35
         fig, ax = plt.subplots(figsize=(8,5))
         ax.set_ylim(-40, -20)
         ax.set_xlim(np.min(grid), np.max(grid))
         lb = 'initial condition'
         ax.plot(grid, w, color=plt.cm.jet(0), lw=2, alpha=0.6, label=lb)
         for i in range(n):
             w = ddp.bellman_operator(w)
             ax.plot(grid, w, color=plt.cm.jet(i / n), lw=2, alpha=0.6)
         lb = 'true value function'
         ax.plot(grid, v_star(grid), 'k-', lw=2, alpha=0.8, label=lb)
         ax.legend(loc='upper left')

         plt.show()
```

We next plot the consumption policies along with the value iteration

```
In [38]: w = 5 * u(grid) - 25           # Initial condition

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

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

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

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

```
    Iteration    Distance       Elapsed (seconds)
--------------------------------------------
1            5.518e+00      2.359e-03
2            4.070e+00      3.584e-03
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
1            5.518e+00      9.956e-04
2            4.070e+00      1.970e-03
3            3.866e+00      2.903e-03
4            3.673e+00      3.735e-03
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
1            5.518e+00      1.229e-03
2            4.070e+00      2.208e-03
3            3.866e+00      3.196e-03
4            3.673e+00      4.297e-03
5            3.489e+00      5.317e-03
6            3.315e+00      6.448e-03
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/quantecon/compute_fp.py:151:
RuntimeWarning: max_iter attained before convergence in compute_fixed_point
  warnings.warn(_non_convergence_msg, RuntimeWarning)
```

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 $k_0 = 0.1$.

```
In [39]: 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()
```

51.8 Appendix: Algorithms

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

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

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

51.8.1 Value Iteration

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

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

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

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

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

51.8.2 Policy Iteration

The `DiscreteDP` policy iteration method runs as follows

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

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

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

51.8.3 Modified Policy Iteration

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

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

- Span is defined by $\text{span}(z) = \max(z) - \min(z)$.

1. 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.

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

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

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

See also the documentation for `DiscreteDP`.

Part VII

LQ Control

Chapter 52

LQ Control: Foundations

52.1 Contents

- Overview 52.2
- Introduction 52.3
- Optimality – Finite Horizon 52.4
- Implementation 52.5
- Extensions and Comments 52.6
- Further Applications 52.7
- Exercises 52.8
- Solutions 52.9

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

```
In [1]: !pip install --upgrade quantecon
```

52.2 Overview

Linear quadratic (LQ) control refers to a class of dynamic optimization problems that have found applications in almost every scientific field.

This lecture provides an introduction to LQ control and its economic applications.

As we will see, LQ systems have a simple structure that makes them an excellent workhorse for a wide variety of economic problems.

Moreover, while the linear-quadratic structure is restrictive, it is in fact far more flexible than it may appear initially.

These themes appear repeatedly below.

Mathematically, LQ control problems are closely related to the Kalman filter

- Recursive formulations of linear-quadratic control problems and Kalman filtering problems both involve matrix **Riccati equations**.
- Classical formulations of linear control and linear filtering problems make use of similar matrix decompositions (see for example this lecture and this lecture).

In reading what follows, it will be useful to have some familiarity with

- matrix manipulations

- vectors of random variables
- dynamic programming and the Bellman equation (see for example this lecture and this lecture)

For additional reading on LQ control, see, for example,

- [108], chapter 5
- [71], chapter 4
- [84], section 3.5

In order to focus on computation, we leave longer proofs to these sources (while trying to provide as much intuition as possible).

Let's start with some imports:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        from quantecon import LQ
```

52.3 Introduction

The "linear" part of LQ is a linear law of motion for the state, while the "quadratic" part refers to preferences.

Let's begin with the former, move on to the latter, and then put them together into an optimization problem.

52.3.1 The Law of Motion

Let x_t be a vector describing the state of some economic system.

Suppose that x_t follows a linear law of motion given by

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \qquad t = 0, 1, 2, ... \tag{1}$$

Here

- u_t is a "control" vector, incorporating choices available to a decision-maker confronting the current state x_t
- $\{w_t\}$ is an uncorrelated zero mean shock process satisfying $\mathbb{E}w_t w_t' = I$, where the right-hand side is the identity matrix

Regarding the dimensions

- x_t is $n \times 1$, A is $n \times n$
- u_t is $k \times 1$, B is $n \times k$
- w_t is $j \times 1$, C is $n \times j$

Example 1

Consider a household budget constraint given by

$$a_{t+1} + c_t = (1+r)a_t + y_t$$

Here a_t is assets, r is a fixed interest rate, c_t is current consumption, and y_t is current non-financial income.

If we suppose that $\{y_t\}$ is serially uncorrelated and $N(0, \sigma^2)$, then, taking $\{w_t\}$ to be standard normal, we can write the system as

$$a_{t+1} = (1+r)a_t - c_t + \sigma w_{t+1}$$

This is clearly a special case of (1), with assets being the state and consumption being the control.

Example 2

One unrealistic feature of the previous model is that non-financial income has a zero mean and is often negative.

This can easily be overcome by adding a sufficiently large mean.

Hence in this example, we take $y_t = \sigma w_{t+1} + \mu$ for some positive real number μ.

Another alteration that's useful to introduce (we'll see why soon) is to change the control variable from consumption to the deviation of consumption from some "ideal" quantity $\bar{c}$.

(Most parameterizations will be such that $\bar{c}$ is large relative to the amount of consumption that is attainable in each period, and hence the household wants to increase consumption)

For this reason, we now take our control to be $u_t := c_t - \bar{c}$.

In terms of these variables, the budget constraint $a_{t+1} = (1+r)a_t - c_t + y_t$ becomes

$$a_{t+1} = (1+r)a_t - u_t - \bar{c} + \sigma w_{t+1} + \mu \tag{2}$$

How can we write this new system in the form of equation (1)?

If, as in the previous example, we take a_t as the state, then we run into a problem: the law of motion contains some constant terms on the right-hand side.

This means that we are dealing with an *affine* function, not a linear one (recall this discussion).

Fortunately, we can easily circumvent this problem by adding an extra state variable.

In particular, if we write

$$\begin{pmatrix} a_{t+1} \\ 1 \end{pmatrix} = \begin{pmatrix} 1+r & -\bar{c}+\mu \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_t \\ 1 \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix} u_t + \begin{pmatrix} \sigma \\ 0 \end{pmatrix} w_{t+1} \tag{3}$$

then the first row is equivalent to (2).

Moreover, the model is now linear and can be written in the form of (1) by setting

$$x_t := \begin{pmatrix} a_t \\ 1 \end{pmatrix}, \quad A := \begin{pmatrix} 1+r & -\bar{c}+\mu \\ 0 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \end{pmatrix} \tag{4}$$

In effect, we've bought ourselves linearity by adding another state.

52.3.2 Preferences

In the LQ model, the aim is to minimize flow of losses, where time-t loss is given by the quadratic expression

$$x_t' R x_t + u_t' Q u_t \tag{5}$$

Here

- R is assumed to be $n \times n$, symmetric and nonnegative definite.
- Q is assumed to be $k \times k$, symmetric and positive definite.

> **Note**
>
> In fact, for many economic problems, the definiteness conditions on R and Q can be relaxed. It is sufficient that certain submatrices of R and Q be nonnegative definite. See [71] for details.

Example 1

A very simple example that satisfies these assumptions is to take R and Q to be identity matrices so that current loss is

$$x_t' I x_t + u_t' I u_t = \|x_t\|^2 + \|u_t\|^2$$

Thus, for both the state and the control, loss is measured as squared distance from the origin.

(In fact, the general case (5) can also be understood in this way, but with R and Q identifying other – non-Euclidean – notions of "distance" from the zero vector).

Intuitively, we can often think of the state x_t as representing deviation from a target, such as

- deviation of inflation from some target level
- deviation of a firm's capital stock from some desired quantity

The aim is to put the state close to the target, while using controls parsimoniously.

Example 2

In the household problem studied above, setting $R = 0$ and $Q = 1$ yields preferences

$$x_t' R x_t + u_t' Q u_t = u_t^2 = (c_t - \bar{c})^2$$

Under this specification, the household's current loss is the squared deviation of consumption from the ideal level $\bar{c}$.

52.4 Optimality – Finite Horizon

Let's now be precise about the optimization problem we wish to consider, and look at how to solve it.

52.4.1 The Objective

We will begin with the finite horizon case, with terminal time $T \in \mathbb{N}$.

In this case, the aim is to choose a sequence of controls $\{u_0, \dots, u_{T-1}\}$ to minimize the objective

$$\mathbb{E}\left\{\sum_{t=0}^{T-1} \beta^t (x_t' R x_t + u_t' Q u_t) + \beta^T x_T' R_f x_T\right\} \tag{6}$$

subject to the law of motion (1) and initial state x_0.

The new objects introduced here are β and the matrix R_f.

The scalar β is the discount factor, while $x' R_f x$ gives terminal loss associated with state x.

Comments:

- We assume R_f to be $n \times n$, symmetric and nonnegative definite.
- We allow $\beta = 1$, and hence include the undiscounted case.
- x_0 may itself be random, in which case we require it to be independent of the shock sequence $w_1, \dots, w_T$.

52.4.2 Information

There's one constraint we've neglected to mention so far, which is that the decision-maker who solves this LQ problem knows only the present and the past, not the future.

To clarify this point, consider the sequence of controls $\{u_0, \dots, u_{T-1}\}$.

When choosing these controls, the decision-maker is permitted to take into account the effects of the shocks $\{w_1, \dots, w_T\}$ on the system.

However, it is typically assumed — and will be assumed here — that the time-t control u_t can be made with knowledge of past and present shocks only.

The fancy measure-theoretic way of saying this is that u_t must be measurable with respect to the σ-algebra generated by $x_0, w_1, w_2, \dots, w_t$.

This is in fact equivalent to stating that u_t can be written in the form $u_t = g_t(x_0, w_1, w_2, \dots, w_t)$ for some Borel measurable function g_t.

(Just about every function that's useful for applications is Borel measurable, so, for the purposes of intuition, you can read that last phrase as "for some function g_t")

Now note that x_t will ultimately depend on the realizations of $x_0, w_1, w_2, \dots, w_t$.

In fact, it turns out that x_t summarizes all the information about these historical shocks that the decision-maker needs to set controls optimally.

More precisely, it can be shown that any optimal control u_t can always be written as a function of the current state alone.

Hence in what follows we restrict attention to control policies (i.e., functions) of the form $u_t = g_t(x_t)$.

Actually, the preceding discussion applies to all standard dynamic programming problems.

What's special about the LQ case is that – as we shall soon see — the optimal u_t turns out to be a linear function of x_t.

52.4.3 Solution

To solve the finite horizon LQ problem we can use a dynamic programming strategy based on backward induction that is conceptually similar to the approach adopted in this lecture.

For reasons that will soon become clear, we first introduce the notation $J_T(x) = x' R_f x$.

Now consider the problem of the decision-maker in the second to last period.

In particular, let the time be $T-1$, and suppose that the state is x_{T-1}.

The decision-maker must trade-off current and (discounted) final losses, and hence solves

$$\min_u \{x'_{T-1} R x_{T-1} + u' Q u + \beta \, \mathbb{E} J_T(A x_{T-1} + B u + C w_T)\}$$

At this stage, it is convenient to define the function

$$J_{T-1}(x) = \min_u \{x' R x + u' Q u + \beta \, \mathbb{E} J_T(A x + B u + C w_T)\} \tag{7}$$

The function J_{T-1} will be called the $T-1$ value function, and $J_{T-1}(x)$ can be thought of as representing total "loss-to-go" from state x at time $T-1$ when the decision-maker behaves optimally.

Now let's step back to $T-2$.

For a decision-maker at $T-2$, the value $J_{T-1}(x)$ plays a role analogous to that played by the terminal loss $J_T(x) = x' R_f x$ for the decision-maker at $T-1$.

That is, $J_{T-1}(x)$ summarizes the future loss associated with moving to state x.

The decision-maker chooses her control u to trade off current loss against future loss, where

- the next period state is $x_{T-1} = A x_{T-2} + B u + C w_{T-1}$, and hence depends on the choice of current control.
- the "cost" of landing in state x_{T-1} is $J_{T-1}(x_{T-1})$.

Her problem is therefore

$$\min_u \{x'_{T-2} R x_{T-2} + u' Q u + \beta \, \mathbb{E} J_{T-1}(A x_{T-2} + B u + C w_{T-1})\}$$

Letting

$$J_{T-2}(x) = \min_u \{x' R x + u' Q u + \beta \, \mathbb{E} J_{T-1}(A x + B u + C w_{T-1})\}$$

the pattern for backward induction is now clear.

In particular, we define a sequence of value functions $\{J_0, \ldots, J_T\}$ via

$$J_{t-1}(x) = \min_u \{x'Rx + u'Qu + \beta\,\mathbb{E}J_t(Ax + Bu + Cw_t)\} \quad \text{and} \quad J_T(x) = x'R_f x$$

The first equality is the Bellman equation from dynamic programming theory specialized to the finite horizon LQ problem.

Now that we have $\{J_0, \dots, J_T\}$, we can obtain the optimal controls.

As a first step, let's find out what the value functions look like.

It turns out that every J_t has the form $J_t(x) = x'P_t x + d_t$ where P_t is a $n \times n$ matrix and d_t is a constant.

We can show this by induction, starting from $P_T := R_f$ and $d_T = 0$.

Using this notation, (7) becomes

$$J_{T-1}(x) = \min_u \{x'Rx + u'Qu + \beta\,\mathbb{E}(Ax + Bu + Cw_T)'P_T(Ax + Bu + Cw_T)\} \tag{8}$$

To obtain the minimizer, we can take the derivative of the r.h.s. with respect to u and set it equal to zero.

Applying the relevant rules of matrix calculus, this gives

$$u = -(Q + \beta B'P_T B)^{-1}\beta B'P_T Ax \tag{9}$$

Plugging this back into (8) and rearranging yields

$$J_{T-1}(x) = x'P_{T-1}x + d_{T-1}$$

where

$$P_{T-1} = R - \beta^2 A'P_T B(Q + \beta B'P_T B)^{-1}B'P_T A + \beta A'P_T A \tag{10}$$

and

$$d_{T-1} := \beta\,\mathrm{trace}(C'P_T C) \tag{11}$$

(The algebra is a good exercise — we'll leave it up to you)

If we continue working backwards in this manner, it soon becomes clear that $J_t(x) = x'P_t x + d_t$ as claimed, where $\{P_t\}$ and $\{d_t\}$ satisfy the recursions

$$P_{t-1} = R - \beta^2 A'P_t B(Q + \beta B'P_t B)^{-1}B'P_t A + \beta A'P_t A \quad \text{with} \quad P_T = R_f \tag{12}$$

and

$$d_{t-1} = \beta(d_t + \mathrm{trace}(C'P_t C)) \quad \text{with} \quad d_T = 0 \tag{13}$$

Recalling (9), the minimizers from these backward steps are

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B'P_{t+1}B)^{-1}\beta B'P_{t+1}A \tag{14}$$

These are the linear optimal control policies we discussed above.

In particular, the sequence of controls given by (14) and (1) solves our finite horizon LQ problem.

Rephrasing this more precisely, the sequence $u_0, \dots, u_{T-1}$ given by

$$u_t = -F_t x_t \quad \text{with} \quad x_{t+1} = (A - BF_t)x_t + Cw_{t+1} \tag{15}$$

for $t = 0, \dots, T-1$ attains the minimum of (6) subject to our constraints.

52.5 Implementation

We will use code from lqcontrol.py in QuantEcon.py to solve finite and infinite horizon linear quadratic control problems.

In the module, the various updating, simulation and fixed point methods are wrapped in a class called `LQ`, which includes

- Instance data:
  - The required parameters Q, R, A, B and optional parameters C, β, T, R_f, N specifying a given LQ model
    - set T and R_f to `None` in the infinite horizon case
    - set `C = None` (or zero) in the deterministic case
  - the value function and policy data
    - d_t, P_t, F_t in the finite horizon case
    - d, P, F in the infinite horizon case
- Methods:
  - `update_values` — shifts d_t, P_t, F_t to their $t-1$ values via (12), (13) and (14)
  - `stationary_values` — computes P, d, F in the infinite horizon case
  - `compute_sequence` —- simulates the dynamics of x_t, u_t, w_t given x_0 and assuming standard normal shocks

52.5.1 An Application

Early Keynesian models assumed that households have a constant marginal propensity to consume from current income.

Data contradicted the constancy of the marginal propensity to consume.

In response, Milton Friedman, Franco Modigliani and others built models based on a consumer's preference for an intertemporally smooth consumption stream.

(See, for example, [56] or [119])

One property of those models is that households purchase and sell financial assets to make consumption streams smoother than income streams.

The household savings problem outlined above captures these ideas.

The optimization problem for the household is to choose a consumption sequence in order to minimize

$$\mathbb{E}\left\{\sum_{t=0}^{T-1}\beta^t(c_t-\bar{c})^2+\beta^T q a_T^2\right\} \tag{16}$$

subject to the sequence of budget constraints $a_{t+1}=(1+r)a_t-c_t+y_t,\ t\geq 0$.

Here q is a large positive constant, the role of which is to induce the consumer to target zero debt at the end of her life.

(Without such a constraint, the optimal choice is to choose $c_t=\bar{c}$ in each period, letting assets adjust accordingly)

As before we set $y_t=\sigma w_{t+1}+\mu$ and $u_t:=c_t-\bar{c}$, after which the constraint can be written as in (2).

We saw how this constraint could be manipulated into the LQ formulation $x_{t+1}=Ax_t+Bu_t+Cw_{t+1}$ by setting $x_t=(a_t\ 1)'$ and using the definitions in (4).

To match with this state and control, the objective function (16) can be written in the form of (6) by choosing

$$Q:=1,\quad R:=\begin{pmatrix}0&0\\0&0\end{pmatrix},\quad\text{and}\quad R_f:=\begin{pmatrix}q&0\\0&0\end{pmatrix}$$

Now that the problem is expressed in LQ form, we can proceed to the solution by applying (12) and (14).

After generating shocks $w_1,\dots,w_T$, the dynamics for assets and consumption can be simulated via (15).

The following figure was computed using $r=0.05,\beta=1/(1+r),\bar{c}=2,\mu=1,\sigma=0.25,T=45$ and $q=10^6$.

The shocks $\{w_t\}$ were taken to be IID and standard normal.

In [3]:
```
# Model parameters
r = 0.05
β = 1/(1 + r)
T = 45
c_bar = 2
σ = 0.25
μ = 1
q = 1e6

# Formulate as an LQ problem
Q = 1
R = np.zeros((2, 2))
Rf = np.zeros((2, 2))
Rf[0, 0] = q
A = [[1 + r, -c_bar + μ],
     [0,              1]]
B = [[-1],
     [ 0]]
C = [[σ],
     [0]]

# Compute solutions and simulate
lq = LQ(Q, R, A, B, C, beta=β, T=T, Rf=Rf)
```

```
x0 = (0, 1)
xp, up, wp = lq.compute_sequence(x0)

# Convert back to assets, consumption and income
assets = xp[0, :]            # a_t
c = up.flatten() + c_bar     # c_t
income = σ * wp[0, 1:] + μ   # y_t

# Plot results
n_rows = 2
fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))

plt.subplots_adjust(hspace=0.5)

bbox = (0., 1.02, 1., .102)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.7}

axes[0].plot(list(range(1, T+1)), income, 'g-', label="non-financial
↪income",
             **p_args)
axes[0].plot(list(range(T)), c, 'k-', label="consumption", **p_args)

axes[1].plot(list(range(1, T+1)), np.cumsum(income - μ), 'r-',
             label="cumulative unanticipated income", **p_args)
axes[1].plot(list(range(T+1)), assets, 'b-', label="assets", **p_args)
axes[1].plot(list(range(T)), np.zeros(T), 'k-')

for ax in axes:
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=2, **legend_args)

plt.show()
```

The top panel shows the time path of consumption c_t and income y_t in the simulation.

As anticipated by the discussion on consumption smoothing, the time path of consumption is much smoother than that for income.

(But note that consumption becomes more irregular towards the end of life, when the zero final asset requirement impinges more on consumption choices).

The second panel in the figure shows that the time path of assets a_t is closely correlated with cumulative unanticipated income, where the latter is defined as

$$z_t := \sum_{j=0}^{t} \sigma w_t$$

A key message is that unanticipated windfall gains are saved rather than consumed, while unanticipated negative shocks are met by reducing assets.

(Again, this relationship breaks down towards the end of life due to the zero final asset requirement)

These results are relatively robust to changes in parameters.

For example, let's increase β from $1/(1+r) \approx 0.952$ to 0.96 while keeping other parameters fixed.

This consumer is slightly more patient than the last one, and hence puts relatively more weight on later consumption values.

In [4]:
```
# Compute solutions and simulate
lq = LQ(Q, R, A, B, C, beta=0.96, T=T, Rf=Rf)
x0 = (0, 1)
xp, up, wp = lq.compute_sequence(x0)

# Convert back to assets, consumption and income
assets = xp[0, :]            # a_t
c = up.flatten() + c_bar     # c_t
income = σ * wp[0, 1:] + μ   # y_t

# Plot results
n_rows = 2
fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))

plt.subplots_adjust(hspace=0.5)

bbox = (0., 1.02, 1., .102)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.7}

axes[0].plot(list(range(1, T+1)), income, 'g-', label="non-financial
↪income",
                 **p_args)
axes[0].plot(list(range(T)), c, 'k-', label="consumption", **p_args)

axes[1].plot(list(range(1, T+1)), np.cumsum(income - μ), 'r-',
             label="cumulative unanticipated income", **p_args)
axes[1].plot(list(range(T+1)), assets, 'b-', label="assets", **p_args)
axes[1].plot(list(range(T)), np.zeros(T), 'k-')

for ax in axes:
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=2, **legend_args)

plt.show()
```

We now have a slowly rising consumption stream and a hump-shaped build-up of assets in the middle periods to fund rising consumption.

However, the essential features are the same: consumption is smooth relative to income, and assets are strongly positively correlated with cumulative unanticipated income.

52.6 Extensions and Comments

Let's now consider a number of standard extensions to the LQ problem treated above.

52.6.1 Time-Varying Parameters

In some settings, it can be desirable to allow A, B, C, R and Q to depend on t.

For the sake of simplicity, we've chosen not to treat this extension in our implementation given below.

However, the loss of generality is not as large as you might first imagine.

In fact, we can tackle many models with time-varying parameters by suitable choice of state variables.

One illustration is given below.

For further examples and a more systematic treatment, see [72], section 2.4.

52.6.2 Adding a Cross-Product Term

In some LQ problems, preferences include a cross-product term $u_t' N x_t$, so that the objective function becomes

$$\mathbb{E}\left\{\sum_{t=0}^{T-1} \beta^t (x_t' R x_t + u_t' Q u_t + 2 u_t' N x_t) + \beta^T x_T' R_f x_T\right\} \tag{17}$$

Our results extend to this case in a straightforward way.

The sequence $\{P_t\}$ from (12) becomes

$$P_{t-1} = R - (\beta B' P_t A + N)'(Q + \beta B' P_t B)^{-1}(\beta B' P_t A + N) + \beta A' P_t A \quad \text{with} \quad P_T = R_f \tag{18}$$

The policies in (14) are modified to

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B' P_{t+1} B)^{-1}(\beta B' P_{t+1} A + N) \tag{19}$$

The sequence $\{d_t\}$ is unchanged from (13).

We leave interested readers to confirm these results (the calculations are long but not overly difficult).

52.6.3 Infinite Horizon

Finally, we consider the infinite horizon case, with cross-product term, unchanged dynamics and objective function given by

$$\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t (x_t' R x_t + u_t' Q u_t + 2 u_t' N x_t)\right\} \tag{20}$$

In the infinite horizon case, optimal policies can depend on time only if time itself is a component of the state vector x_t.

In other words, there exists a fixed matrix F such that $u_t = -F x_t$ for all t.

That decision rules are constant over time is intuitive — after all, the decision-maker faces the same infinite horizon at every stage, with only the current state changing.

Not surprisingly, P and d are also constant.

The stationary matrix P is the solution to the discrete-time algebraic Riccati equation.

$$P = R - (\beta B' P A + N)'(Q + \beta B' P B)^{-1}(\beta B' P A + N) + \beta A' P A \tag{21}$$

Equation (21) is also called the *LQ Bellman equation*, and the map that sends a given P into the right-hand side of (21) is called the *LQ Bellman operator*.

The stationary optimal policy for this model is

$$u = -F x \quad \text{where} \quad F = (Q + \beta B' P B)^{-1}(\beta B' P A + N) \tag{22}$$

The sequence $\{d_t\}$ from (13) is replaced by the constant value

$$d := \text{trace}(C'PC)\frac{\beta}{1-\beta} \tag{23}$$

The state evolves according to the time-homogeneous process $x_{t+1} = (A - BF)x_t + Cw_{t+1}$.

An example infinite horizon problem is treated below.

52.6.4 Certainty Equivalence

Linear quadratic control problems of the class discussed above have the property of *certainty equivalence*.

By this, we mean that the optimal policy F is not affected by the parameters in C, which specify the shock process.

This can be confirmed by inspecting (22) or (19).

It follows that we can ignore uncertainty when solving for optimal behavior, and plug it back in when examining optimal state dynamics.

52.7 Further Applications

52.7.1 Application 1: Age-Dependent Income Process

Previously we studied a permanent income model that generated consumption smoothing.

One unrealistic feature of that model is the assumption that the mean of the random income process does not depend on the consumer's age.

A more realistic income profile is one that rises in early working life, peaks towards the middle and maybe declines toward the end of working life and falls more during retirement.

In this section, we will model this rise and fall as a symmetric inverted "U" using a polynomial in age.

As before, the consumer seeks to minimize

$$\mathbb{E}\left\{\sum_{t=0}^{T-1}\beta^t(c_t-\bar{c})^2+\beta^T q a_T^2\right\} \tag{24}$$

subject to $a_{t+1} = (1+r)a_t - c_t + y_t,\ t \geq 0$.

For income we now take $y_t = p(t) + \sigma w_{t+1}$ where $p(t) := m_0 + m_1 t + m_2 t^2$.

(In the next section we employ some tricks to implement a more sophisticated model)

The coefficients m_0, m_1, m_2 are chosen such that $p(0) = 0, p(T/2) = \mu$, and $p(T) = 0$.

You can confirm that the specification $m_0 = 0, m_1 = T\mu/(T/2)^2, m_2 = -\mu/(T/2)^2$ satisfies these constraints.

To put this into an LQ setting, consider the budget constraint, which becomes

$$a_{t+1} = (1+r)a_t - u_t - \bar{c} + m_1 t + m_2 t^2 + \sigma w_{t+1} \tag{25}$$

The fact that a_{t+1} is a linear function of $(a_t, 1, t, t^2)$ suggests taking these four variables as the state vector x_t.

Once a good choice of state and control (recall $u_t = c_t - \bar{c}$) has been made, the remaining specifications fall into place relatively easily.

Thus, for the dynamics we set

$$x_t := \begin{pmatrix} a_t \\ 1 \\ t \\ t^2 \end{pmatrix}, \quad A := \begin{pmatrix} 1+r & -\bar{c} & m_1 & m_2 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{26}$$

If you expand the expression $x_{t+1} = Ax_t + Bu_t + Cw_{t+1}$ using this specification, you will find that assets follow (25) as desired and that the other state variables also update appropriately.

To implement preference specification (24) we take

$$Q := 1, \quad R := \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad R_f := \begin{pmatrix} q & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{27}$$

The next figure shows a simulation of consumption and assets computed using the `compute_sequence` method of `lqcontrol.py` with initial assets set to zero.

Once again, smooth consumption is a dominant feature of the sample paths.

The asset path exhibits dynamics consistent with standard life cycle theory.

Exercise 1 gives the full set of parameters used here and asks you to replicate the figure.

52.7.2 Application 2: A Permanent Income Model with Retirement

In the previous application, we generated income dynamics with an inverted U shape using polynomials and placed them in an LQ framework.

It is arguably the case that this income process still contains unrealistic features.

A more common earning profile is where

1. income grows over working life, fluctuating around an increasing trend, with growth flattening off in later years
2. retirement follows, with lower but relatively stable (non-financial) income

Letting K be the retirement date, we can express these income dynamics by

$$y_t = \begin{cases} p(t) + \sigma w_{t+1} & \text{if } t \leq K \\ s & \text{otherwise} \end{cases} \tag{28}$$

Here

- $p(t) := m_1 t + m_2 t^2$ with the coefficients m_1, m_2 chosen such that $p(K) = \mu$ and $p(0) = p(2K) = 0$
- s is retirement income

We suppose that preferences are unchanged and given by (16).

The budget constraint is also unchanged and given by $a_{t+1} = (1+r)a_t - c_t + y_t$.

Our aim is to solve this problem and simulate paths using the LQ techniques described in this lecture.

In fact, this is a nontrivial problem, as the kink in the dynamics (28) at K makes it very difficult to express the law of motion as a fixed-coefficient linear system.

However, we can still use our LQ methods here by suitably linking two-component LQ problems.

These two LQ problems describe the consumer's behavior during her working life (`lq_working`) and retirement (`lq_retired`).

(This is possible because, in the two separate periods of life, the respective income processes [polynomial trend and constant] each fit the LQ framework)

The basic idea is that although the whole problem is not a single time-invariant LQ problem, it is still a dynamic programming problem, and hence we can use appropriate Bellman equations at every stage.

Based on this logic, we can

1. solve `lq_retired` by the usual backward induction procedure, iterating back to the start of retirement.

2. take the start-of-retirement value function generated by this process, and use it as the terminal condition R_f to feed into the `lq_working` specification.

3. solve `lq_working` by backward induction from this choice of R_f, iterating back to the start of working life.

This process gives the entire life-time sequence of value functions and optimal policies.

The next figure shows one simulation based on this procedure.

The full set of parameters used in the simulation is discussed in Exercise 2, where you are asked to replicate the figure.

Once again, the dominant feature observable in the simulation is consumption smoothing.

The asset path fits well with standard life cycle theory, with dissaving early in life followed by later saving.

Assets peak at retirement and subsequently decline.

52.7.3 Application 3: Monopoly with Adjustment Costs

Consider a monopolist facing stochastic inverse demand function

$$p_t = a_0 - a_1 q_t + d_t$$

Here q_t is output, and the demand shock d_t follows

$$d_{t+1} = \rho d_t + \sigma w_{t+1}$$

where $\{w_t\}$ is IID and standard normal.

The monopolist maximizes the expected discounted sum of present and future profits

$$\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t \pi_t\right\} \quad \text{where} \quad \pi_t := p_t q_t - c q_t - \gamma(q_{t+1} - q_t)^2 \tag{29}$$

Here

- $\gamma(q_{t+1} - q_t)^2$ represents adjustment costs
- c is average cost of production

This can be formulated as an LQ problem and then solved and simulated, but first let's study the problem and try to get some intuition.

One way to start thinking about the problem is to consider what would happen if $\gamma = 0$.

Without adjustment costs there is no intertemporal trade-off, so the monopolist will choose output to maximize current profit in each period.

It's not difficult to show that profit-maximizing output is

$$\bar{q}_t := \frac{a_0 - c + d_t}{2a_1}$$

In light of this discussion, what we might expect for general γ is that

- if γ is close to zero, then q_t will track the time path of $\bar{q}_t$ relatively closely.
- if γ is larger, then q_t will be smoother than $\bar{q}_t$, as the monopolist seeks to avoid adjustment costs.

This intuition turns out to be correct.

The following figures show simulations produced by solving the corresponding LQ problem.

The only difference in parameters across the figures is the size of γ

$\bar{q}_t$
q_t
dynamics with $\gamma = 1$
3.75
3.50
3.25
3.00
2.75
2.50
2.25
2.00
0
20
40
60
80
100
120
140
Time

$\bar{q}_t$
q_t
dynamics with $\gamma = 10$
4.0
3.5
3.0
2.5
2.0
0
20
40
60
80
100
120
140
Time

To produce these figures we converted the monopolist problem into an LQ problem.

The key to this conversion is to choose the right state — which can be a bit of an art.

Here we take $x_t = (\bar{q}_t \;\; q_t \;\; 1)'$, while the control is chosen as $u_t = q_{t+1} - q_t$.

We also manipulated the profit function slightly.

In (29), current profits are $\pi_t := p_t q_t - c q_t - \gamma(q_{t+1} - q_t)^2$.

Let's now replace π_t in (29) with $\hat{\pi}_t := \pi_t - a_1 \bar{q}_t^2$.

This makes no difference to the solution, since $a_1 \bar{q}_t^2$ does not depend on the controls.

(In fact, we are just adding a constant term to (29), and optimizers are not affected by constant terms)

The reason for making this substitution is that, as you will be able to verify, $\hat{\pi}_t$ reduces to the simple quadratic

$$\hat{\pi}_t = -a_1(q_t - \bar{q}_t)^2 - \gamma u_t^2$$

After negation to convert to a minimization problem, the objective becomes

$$\min \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left\{ a_1(q_t - \bar{q}_t)^2 + \gamma u_t^2 \right\} \tag{30}$$

It's now relatively straightforward to find R and Q such that (30) can be written as (20).

Furthermore, the matrices A, B and C from (1) can be found by writing down the dynamics of each element of the state.

Exercise 3 asks you to complete this process, and reproduce the preceding figures.

52.8 Exercises

52.8.1 Exercise 1

Replicate the figure with polynomial income shown above.

The parameters are $r = 0.05, \beta = 1/(1+r), \bar{c} = 1.5, \mu = 2, \sigma = 0.15, T = 50$ and $q = 10^4$.

52.8.2 Exercise 2

Replicate the figure on work and retirement shown above.

The parameters are $r = 0.05, \beta = 1/(1+r), \bar{c} = 4, \mu = 4, \sigma = 0.35, K = 40, T = 60, s = 1$ and $q = 10^4$.

To understand the overall procedure, carefully read the section containing that figure.

Some hints are as follows:

First, in order to make our approach work, we must ensure that both LQ problems have the same state variables and control.

As with previous applications, the control can be set to $u_t = c_t - \bar{c}$.

For `lq_working`, x_t, A, B, C can be chosen as in (26).

- Recall that m_1, m_2 are chosen so that $p(K) = \mu$ and $p(2K) = 0$.

For `lq_retired`, use the same definition of x_t and u_t, but modify A, B, C to correspond to constant income $y_t = s$.

For `lq_retired`, set preferences as in (27).

For `lq_working`, preferences are the same, except that R_f should be replaced by the final value function that emerges from iterating `lq_retired` back to the start of retirement.

With some careful footwork, the simulation can be generated by patching together the simulations from these two separate models.

52.8.3 Exercise 3

Reproduce the figures from the monopolist application given above.

For parameters, use $a_0 = 5, a_1 = 0.5, \sigma = 0.15, \rho = 0.9, \beta = 0.95$ and $c = 2$, while γ varies between 1 and 50 (see figures).

52.9 Solutions

52.9.1 Exercise 1

Here's one solution.

We use some fancy plot commands to get a certain style — feel free to use simpler ones.

The model is an LQ permanent income / life-cycle model with hump-shaped income

$$y_t = m_1 t + m_2 t^2 + \sigma w_{t+1}$$

where $\{w_t\}$ is IID $N(0,1)$ and the coefficients m_1 and m_2 are chosen so that $p(t) = m_1 t + m_2 t^2$ has an inverted U shape with

- $p(0) = 0, p(T/2) = \mu$, and
- $p(T) = 0$

In [5]:
```
# Model parameters
r = 0.05
β = 1/(1 + r)
T = 50
c_bar = 1.5
σ = 0.15
μ = 2
q = 1e4
m1 = T * (μ/(T/2)**2)
m2 = -(μ/(T/2)**2)

# Formulate as an LQ problem
Q = 1
R = np.zeros((4, 4))
Rf = np.zeros((4, 4))
Rf[0, 0] = q
A = [[1 + r, -c_bar, m1, m2],
     [0,          1,  0,  0],
     [0,          1,  1,  0],
     [0,          1,  2,  1]]
B = [[-1],
     [ 0],
     [ 0],
     [ 0]]
C = [[σ],
     [0],
     [0],
     [0]]

# Compute solutions and simulate
lq = LQ(Q, R, A, B, C, beta=β, T=T, Rf=Rf)
x0 = (0, 1, 0, 0)
xp, up, wp = lq.compute_sequence(x0)

# Convert results back to assets, consumption and income
ap = xp[0, :]               # Assets
c = up.flatten() + c_bar    # Consumption
time = np.arange(1, T+1)
income = σ * wp[0, 1:] + m1 * time + m2 * time**2  # Income


# Plot results
n_rows = 2
fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))

plt.subplots_adjust(hspace=0.5)

bbox = (0., 1.02, 1., .102)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
```

```
p_args = {'lw': 2, 'alpha': 0.7}

axes[0].plot(range(1, T+1), income, 'g-', label="non-financial income",
             **p_args)
axes[0].plot(range(T), c, 'k-', label="consumption", **p_args)

axes[1].plot(range(T+1), ap.flatten(), 'b-', label="assets", **p_args)
axes[1].plot(range(T+1), np.zeros(T+1), 'k-')

for ax in axes:
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=2, **legend_args)

plt.show()
```

52.9.2 Exercise 2

This is a permanent income / life-cycle model with polynomial growth in income over working life followed by a fixed retirement income.

The model is solved by combining two LQ programming problems as described in the lecture.

In [6]:
```
# Model parameters
r = 0.05
β = 1/(1 + r)
T = 60
K = 40
c_bar = 4
σ = 0.35
μ = 4
q = 1e4
s = 1
m1 = 2 * μ/K
m2 = -μ/K**2

# Formulate LQ problem 1 (retirement)
Q = 1
R = np.zeros((4, 4))
Rf = np.zeros((4, 4))
Rf[0, 0] = q
A = [[1 + r, s - c_bar, 0, 0],
     [0,             1, 0, 0],
     [0,             1, 1, 0],
     [0,             1, 2, 1]]
B = [[-1],
     [ 0],
     [ 0],
     [ 0]]
C = [[0],
     [0],
     [0],
     [0]]

# Initialize LQ instance for retired agent
lq_retired = LQ(Q, R, A, B, C, beta=β, T=T-K, Rf=Rf)
# Iterate back to start of retirement, record final value function
for i in range(T-K):
    lq_retired.update_values()
Rf2 = lq_retired.P

# Formulate LQ problem 2 (working life)
R = np.zeros((4, 4))
A = [[1 + r, -c_bar, m1, m2],
     [0,          1,  0,  0],
     [0,          1,  1,  0],
     [0,          1,  2,  1]]
B = [[-1],
     [ 0],
     [ 0],
     [ 0]]
C = [[σ],
     [0],
     [0],
     [0]]

# Set up working life LQ instance with terminal Rf from lq_retired
lq_working = LQ(Q, R, A, B, C, beta=β, T=K, Rf=Rf2)

# Simulate working state / control paths
x0 = (0, 1, 0, 0)
```

```
xp_w, up_w, wp_w = lq_working.compute_sequence(x0)
# Simulate retirement paths (note the initial condition)
xp_r, up_r, wp_r = lq_retired.compute_sequence(xp_w[:, K])

# Convert results back to assets, consumption and income
xp = np.column_stack((xp_w, xp_r[:, 1:]))
assets = xp[0, :]                   # Assets

up = np.column_stack((up_w, up_r))
c = up.flatten() + c_bar            # Consumption

time = np.arange(1, K+1)
income_w = σ * wp_w[0, 1:K+1] + m1 * time + m2 * time**2  # Income
income_r = np.ones(T-K) * s
income = np.concatenate((income_w, income_r))

# Plot results
n_rows = 2
fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))

plt.subplots_adjust(hspace=0.5)

bbox = (0., 1.02, 1., .102)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.7}

axes[0].plot(range(1, T+1), income, 'g-', label="non-financial income",
             **p_args)
axes[0].plot(range(T), c, 'k-', label="consumption", **p_args)

axes[1].plot(range(T+1), assets, 'b-', label="assets", **p_args)
axes[1].plot(range(T+1), np.zeros(T+1), 'k-')

for ax in axes:
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=2, **legend_args)

plt.show()
```

52.9.3 Exercise 3

The first task is to find the matrices A, B, C, Q, R that define the LQ problem.

Recall that $x_t = (\bar{q}_t \;\; q_t \;\; 1)'$, while $u_t = q_{t+1} - q_t$.

Letting $m_0 := (a_0 - c)/2a_1$ and $m_1 := 1/2a_1$, we can write $\bar{q}_t = m_0 + m_1 d_t$, and then, with some manipulation

$$\bar{q}_{t+1} = m_0(1-\rho) + \rho\bar{q}_t + m_1\sigma w_{t+1}$$

By our definition of u_t, the dynamics of q_t are $q_{t+1} = q_t + u_t$.

Using these facts you should be able to build the correct A, B, C matrices (and then check them against those found in the solution code below).

Suitable R, Q matrices can be found by inspecting the objective function, which we repeat here for convenience:

$$\min \mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t a_1(q_t - \bar{q}_t)^2 + \gamma u_t^2\right\}$$

Our solution code is

In [7]:
```
# Model parameters
a0 = 5
a1 = 0.5
σ = 0.15
ρ = 0.9
γ = 1
β = 0.95
c = 2
T = 120

# Useful constants
m0 = (a0-c)/(2 * a1)
m1 = 1/(2 * a1)

# Formulate LQ problem
Q = γ
R = [[ a1, -a1,  0],
     [-a1,  a1,  0],
     [  0,   0,  0]]
A = [[ρ, 0, m0 * (1 - ρ)],
     [0, 1,            0],
     [0, 0,            1]]

B = [[0],
     [1],
     [0]]
C = [[m1 * σ],
     [     0],
     [     0]]

lq = LQ(Q, R, A, B, C=C, beta=β)

# Simulate state / control paths
x0 = (m0, 2, 1)
xp, up, wp = lq.compute_sequence(x0, ts_length=150)
q_bar = xp[0, :]
q = xp[1, :]

# Plot simulation results
fig, ax = plt.subplots(figsize=(10, 6.5))

# Some fancy plotting stuff -- simplify if you prefer
bbox = (0., 1.01, 1., .101)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.6}

time = range(len(q))
ax.set(xlabel='Time', xlim=(0, max(time)))
ax.plot(time, q_bar, 'k-', lw=2, alpha=0.6, label=r'$\bar q_t$')
ax.plot(time, q, 'b-', lw=2, alpha=0.6, label='$q_t$')
ax.legend(ncol=2, **legend_args)
s = f'dynamics with $\gamma = {γ}$'
ax.text(max(time) * 0.6, 1 * q_bar.max(), s, fontsize=14)
plt.show()
```

$\bar{q}_t$
q_t
dynamics with $\gamma = 1$
4.0
3.5
3.0
2.5
2.0
0
20
40
60
80
100
120
140
Time

Chapter 53

Optimal Savings I: The Permanent Income Model

53.1 Contents

- Overview 53.2
- The Savings Problem 53.3
- Alternative Representations 53.4
- Two Classic Examples 53.5
- Further Reading 53.6
- Appendix: The Euler Equation 53.7

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

```
In [1]: !pip install --upgrade quantecon
```

53.2 Overview

This lecture describes a rational expectations version of the famous permanent income model of Milton Friedman [56].

Robert Hall cast Friedman's model within a linear-quadratic setting [67].

Like Hall, we formulate an infinite-horizon linear-quadratic savings problem.

We use the model as a vehicle for illustrating

- alternative formulations of the *state* of a dynamic system
- the idea of *cointegration*
- impulse response functions
- the idea that changes in consumption are useful as predictors of movements in income

Background readings on the linear-quadratic-Gaussian permanent income model are Hall's [67] and chapter 2 of [108].

Let's start with some imports

```
In [2]: import matplotlib.pyplot as plt
        %matplotlib inline
```

```
import numpy as np
import random
from numba import njit
```

53.3 The Savings Problem

In this section, we state and solve the savings and consumption problem faced by the consumer.

53.3.1 Preliminaries

We use a class of stochastic processes called martingales.

A discrete-time martingale is a stochastic process (i.e., a sequence of random variables) $\{X_t\}$ with finite mean at each t and satisfying

$$\mathbb{E}_t[X_{t+1}] = X_t, \qquad t = 0, 1, 2, \ldots$$

Here $\mathbb{E}_t := \mathbb{E}[\cdot \mid \mathcal{F}_t]$ is a conditional mathematical expectation conditional on the time t *information set* $\mathcal{F}_t$.

The latter is just a collection of random variables that the modeler declares to be visible at t.

- When not explicitly defined, it is usually understood that $\mathcal{F}_t = \{X_t, X_{t-1}, \ldots, X_0\}$.

Martingales have the feature that the history of past outcomes provides no predictive power for changes between current and future outcomes.

For example, the current wealth of a gambler engaged in a "fair game" has this property.

One common class of martingales is the family of *random walks*.

A **random walk** is a stochastic process $\{X_t\}$ that satisfies

$$X_{t+1} = X_t + w_{t+1}$$

for some IID zero mean *innovation* sequence $\{w_t\}$.

Evidently, X_t can also be expressed as

$$X_t = \sum_{j=1}^{t} w_j + X_0$$

Not every martingale arises as a random walk (see, for example, Wald's martingale).

53.3.2 The Decision Problem

A consumer has preferences over consumption streams that are ordered by the utility functional

$$\mathbb{E}_0 \left[\sum_{t=0}^{\infty} \beta^t u(c_t) \right] \tag{1}$$

where

- $\mathbb{E}_t$ is the mathematical expectation conditioned on the consumer's time t information
- c_t is time t consumption
- u is a strictly concave one-period utility function
- $\beta \in (0, 1)$ is a discount factor

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

$$c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t \quad t \geq 0 \tag{2}$$

Here

- y_t is an exogenous endowment process.
- $r > 0$ is a time-invariant risk-free net interest rate.
- b_t is one-period risk-free debt maturing at t.

The consumer also faces initial conditions b_0 and y_0, which can be fixed or random.

53.3.3 Assumptions

For the remainder of this lecture, we follow Friedman and Hall in assuming that $(1+r)^{-1} = \beta$.

Regarding the endowment process, we assume it has the state-space representation

$$\begin{aligned} z_{t+1} &= Az_t + Cw_{t+1} \\ y_t &= Uz_t \end{aligned} \tag{3}$$

where

- $\{w_t\}$ is an IID vector process with $\mathbb{E}w_t = 0$ and $\mathbb{E}w_t w_t' = I$.
- The spectral radius of A satisfies $\rho(A) < \sqrt{1/\beta}$.
- U is a selection vector that pins down y_t as a particular linear combination of components of z_t.

The restriction on $\rho(A)$ prevents income from growing so fast that discounted geometric sums of some quadratic forms to be described below become infinite.

Regarding preferences, we assume the quadratic utility function

$$u(c_t) = -(c_t - \gamma)^2$$

where γ is a bliss level of consumption

> **Note**
>
> Along with this quadratic utility specification, we allow consumption to be negative. However, by choosing parameters appropriately, we can make the probability that the model generates negative consumption paths over finite time horizons as low as desired.

Finally, we impose the *no Ponzi scheme* condition

$$\mathbb{E}_0 \left[\sum_{t=0}^{\infty} \beta^t b_t^2 \right] < \infty \tag{4}$$

This condition rules out an always-borrow scheme that would allow the consumer to enjoy bliss consumption forever.

53.3.4 First-Order Conditions

First-order conditions for maximizing (1) subject to (2) are

$$\mathbb{E}_t[u'(c_{t+1})] = u'(c_t), \qquad t = 0, 1, \ldots \tag{5}$$

These optimality conditions are also known as *Euler equations*.

If you're not sure where they come from, you can find a proof sketch in the appendix.

With our quadratic preference specification, (5) has the striking implication that consumption follows a martingale:

$$\mathbb{E}_t[c_{t+1}] = c_t \tag{6}$$

(In fact, quadratic preferences are *necessary* for this conclusion Section **??**)

One way to interpret (6) is that consumption will change only when "new information" about permanent income is revealed.

These ideas will be clarified below.

53.3.5 The Optimal Decision Rule

Now let's deduce the optimal decision rule Section **??**

> **Note**
>
> One way to solve the consumer's problem is to apply *dynamic programming* as in this lecture. We do this later. But first we use an alternative approach that is revealing and shows the work that dynamic programming does for us behind the scenes.

In doing so, we need to combine

1. the optimality condition (6)
2. the period-by-period budget constraint (2), and
3. the boundary condition (4)

To accomplish this, observe first that (4) implies $\lim_{t\to\infty} \beta^{\frac{t}{2}} b_{t+1} = 0$.

Using this restriction on the debt path and solving (2) forward yields

$$b_t = \sum_{j=0}^{\infty} \beta^j (y_{t+j} - c_{t+j}) \tag{7}$$

Take conditional expectations on both sides of (7) and use the martingale property of consumption and the *law of iterated expectations* to deduce

$$b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - \frac{c_t}{1-\beta} \tag{8}$$

Expressed in terms of c_t we get

$$c_t = (1-\beta)\left[\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - b_t\right] = \frac{r}{1+r}\left[\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - b_t\right] \tag{9}$$

where the last equality uses $(1+r)\beta = 1$.

These last two equations assert that consumption equals *economic income*

- **financial wealth** equals $-b_t$
- **non-financial wealth** equals $\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}]$
- **total wealth** equals the sum of financial and non-financial wealth
- a **marginal propensity to consume out of total wealth** equals the interest factor $\frac{r}{1+r}$
- **economic income** equals
  - a constant marginal propensity to consume times the sum of non-financial wealth and financial wealth
  - the amount the consumer can consume while leaving its wealth intact

Responding to the State

The *state* vector confronting the consumer at t is $\begin{bmatrix} b_t & z_t \end{bmatrix}$.

Here

- z_t is an *exogenous* component, unaffected by consumer behavior.
- b_t is an *endogenous* component (since it depends on the decision rule).

Note that z_t contains all variables useful for forecasting the consumer's future endowment.

It is plausible that current decisions c_t and b_{t+1} should be expressible as functions of z_t and b_t.

This is indeed the case.

In fact, from this discussion, we see that

$$\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] = \mathbb{E}_t\left[\sum_{j=0}^{\infty} \beta^j y_{t+j}\right] = U(I-\beta A)^{-1} z_t$$

Combining this with (9) gives

$$c_t = \frac{r}{1+r}\left[U(I-\beta A)^{-1} z_t - b_t\right] \tag{10}$$

Using this equality to eliminate c_t in the budget constraint (2) gives

$$\begin{aligned} b_{t+1} &= (1+r)(b_t + c_t - y_t) \\ &= (1+r)b_t + r[U(I-\beta A)^{-1}z_t - b_t] - (1+r)Uz_t \\ &= b_t + U[r(I-\beta A)^{-1} - (1+r)I]z_t \\ &= b_t + U(I-\beta A)^{-1}(A-I)z_t \end{aligned}$$

To get from the second last to the last expression in this chain of equalities is not trivial.

A key is to use the fact that $(1+r)\beta = 1$ and $(I-\beta A)^{-1} = \sum_{j=0}^{\infty}\beta^j A^j$.

We've now successfully written c_t and b_{t+1} as functions of b_t and z_t.

A State-Space Representation

We can summarize our dynamics in the form of a linear state-space system governing consumption, debt and income:

$$\begin{aligned} z_{t+1} &= Az_t + Cw_{t+1} \\ b_{t+1} &= b_t + U[(I-\beta A)^{-1}(A-I)]z_t \\ y_t &= Uz_t \\ c_t &= (1-\beta)[U(I-\beta A)^{-1}z_t - b_t] \end{aligned} \tag{11}$$

To write this more succinctly, let

$$x_t = \begin{bmatrix} z_t \\ b_t \end{bmatrix}, \quad \tilde{A} = \begin{bmatrix} A & 0 \\ U(I-\beta A)^{-1}(A-I) & 1 \end{bmatrix}, \quad \tilde{C} = \begin{bmatrix} C \\ 0 \end{bmatrix}$$

and

$$\tilde{U} = \begin{bmatrix} U & 0 \\ (1-\beta)U(I-\beta A)^{-1} & -(1-\beta) \end{bmatrix}, \quad \tilde{y}_t = \begin{bmatrix} y_t \\ c_t \end{bmatrix}$$

Then we can express equation (11) as

$$\begin{aligned} x_{t+1} &= \tilde{A}x_t + \tilde{C}w_{t+1} \\ \tilde{y}_t &= \tilde{U}x_t \end{aligned} \tag{12}$$

We can use the following formulas from linear state space models to compute population mean $\mu_t = \mathbb{E}x_t$ and covariance $\Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)']$

$$\mu_{t+1} = \tilde{A}\mu_t \quad \text{with} \quad \mu_0 \text{ given} \tag{13}$$

$$\Sigma_{t+1} = \tilde{A}\Sigma_t\tilde{A}' + \tilde{C}\tilde{C}' \quad \text{with} \quad \Sigma_0 \text{ given} \tag{14}$$

We can then compute the mean and covariance of $\tilde{y}_t$ from

$$\begin{aligned} \mu_{y,t} &= \tilde{U}\mu_t \\ \Sigma_{y,t} &= \tilde{U}\Sigma_t\tilde{U}' \end{aligned} \tag{15}$$

A Simple Example with IID Income

To gain some preliminary intuition on the implications of (11), let's look at a highly stylized example where income is just IID.

(Later examples will investigate more realistic income streams)

In particular, let $\{w_t\}_{t=1}^{\infty}$ be IID and scalar standard normal, and let

$$z_t = \begin{bmatrix} z_t^1 \\ 1 \end{bmatrix}, \quad A = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & \mu \end{bmatrix}, \quad C = \begin{bmatrix} \sigma \\ 0 \end{bmatrix}$$

Finally, let $b_0 = z_0^1 = 0$.

Under these assumptions, we have $y_t = \mu + \sigma w_t \sim N(\mu, \sigma^2)$.

Further, if you work through the state space representation, you will see that

$$b_t = -\sigma \sum_{j=1}^{t-1} w_j$$

$$c_t = \mu + (1-\beta)\sigma \sum_{j=1}^{t} w_j$$

Thus income is IID and debt and consumption are both Gaussian random walks.

Defining assets as $-b_t$, we see that assets are just the cumulative sum of unanticipated incomes prior to the present date.

The next figure shows a typical realization with $r = 0.05$, $\mu = 1$, and $\sigma = 0.15$

In [3]:
```
r = 0.05
β = 1 / (1 + r)
σ = 0.15
μ = 1
T = 60

@njit
def time_path(T):
    w = np.random.randn(T+1)  # w_0, w_1, ..., w_T
    w[0] = 0
    b = np.zeros(T+1)
    for t in range(1, T+1):
        b[t] = w[1:t].sum()
    b = -σ * b
    c = μ + (1 - β) * (σ * w - b)
    return w, b, c

w, b, c = time_path(T)

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(μ + σ * w, 'g-', label="Non-financial income")
ax.plot(c, 'k-', label="Consumption")
ax.plot( b, 'b-', label="Debt")
ax.legend(ncol=3, mode='expand', bbox_to_anchor=(0., 1.02, 1., .102))
ax.grid()
```

```
ax.set_xlabel('Time')

plt.show()
```

Observe that consumption is considerably smoother than income.

The figure below shows the consumption paths of 250 consumers with independent income streams

```
In [4]: fig, ax = plt.subplots(figsize=(10, 6))

        b_sum = np.zeros(T+1)
        for i in range(250):
            w, b, c = time_path(T)  # Generate new time path
            rcolor = random.choice(('c', 'g', 'b', 'k'))
            ax.plot(c, color=rcolor, lw=0.8, alpha=0.7)

        ax.grid()
        ax.set(xlabel='Time', ylabel='Consumption')

        plt.show()
```

53.4 Alternative Representations

In this section, we shed more light on the evolution of savings, debt and consumption by representing their dynamics in several different ways.

53.4.1 Hall's Representation

Hall [67] suggested an insightful way to summarize the implications of LQ permanent income theory.

First, to represent the solution for b_t, shift (9) forward one period and eliminate b_{t+1} by using (2) to obtain

$$c_{t+1} = (1-\beta)\sum_{j=0}^{\infty}\beta^j \mathbb{E}_{t+1}[y_{t+j+1}] - (1-\beta)\left[\beta^{-1}(c_t + b_t - y_t)\right]$$

If we add and subtract $\beta^{-1}(1-\beta)\sum_{j=0}^{\infty}\beta^j \mathbb{E}_t y_{t+j}$ from the right side of the preceding equation and rearrange, we obtain

$$c_{t+1} - c_t = (1-\beta)\sum_{j=0}^{\infty}\beta^j \left\{\mathbb{E}_{t+1}[y_{t+j+1}] - \mathbb{E}_t[y_{t+j+1}]\right\} \tag{16}$$

The right side is the time $t+1$ *innovation to the expected present value* of the endowment process $\{y_t\}$.

We can represent the optimal decision rule for (c_t, b_{t+1}) in the form of (16) and (8), which we repeat:

$$b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - \frac{1}{1-\beta} c_t \tag{17}$$

Equation (17) asserts that the consumer's debt due at t equals the expected present value of its endowment minus the expected present value of its consumption stream.

A high debt thus indicates a large expected present value of surpluses $y_t - c_t$.

Recalling again our discussion on forecasting geometric sums, we have

$$\begin{aligned}
\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} &= U(I - \beta A)^{-1} z_t \\
\mathbb{E}_{t+1} \sum_{j=0}^{\infty} \beta^j y_{t+j+1} &= U(I - \beta A)^{-1} z_{t+1} \\
\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j+1} &= U(I - \beta A)^{-1} A z_t
\end{aligned}$$

Using these formulas together with (3) and substituting into (16) and (17) gives the following representation for the consumer's optimum decision rule:

$$\begin{aligned}
c_{t+1} &= c_t + (1-\beta) U (I - \beta A)^{-1} C w_{t+1} \\
b_t &= U(I - \beta A)^{-1} z_t - \frac{1}{1-\beta} c_t \\
y_t &= U z_t \\
z_{t+1} &= A z_t + C w_{t+1}
\end{aligned} \tag{18}$$

Representation (18) makes clear that

- The state can be taken as (c_t, z_t).
  - The endogenous part is c_t and the exogenous part is z_t.
  - Debt b_t has disappeared as a component of the state because it is encoded in c_t.
- Consumption is a random walk with innovation $(1-\beta) U (I - \beta A)^{-1} C w_{t+1}$.
  - This is a more explicit representation of the martingale result in (6).

53.4.2 Cointegration

Representation (18) reveals that the joint process $\{c_t, b_t\}$ possesses the property that Engle and Granger [51] called cointegration.

Cointegration is a tool that allows us to apply powerful results from the theory of stationary stochastic processes to (certain transformations of) nonstationary models.

To apply cointegration in the present context, suppose that z_t is asymptotically stationary Section **??**.

Despite this, both c_t and b_t will be non-stationary because they have unit roots (see (11) for b_t).

Nevertheless, there is a linear combination of c_t, b_t that *is* asymptotically stationary.

In particular, from the second equality in (18) we have

$$(1-\beta)b_t + c_t = (1-\beta)U(I-\beta A)^{-1}z_t \tag{19}$$

Hence the linear combination $(1-\beta)b_t + c_t$ is asymptotically stationary.

Accordingly, Granger and Engle would call $\begin{bmatrix}(1-\beta) & 1\end{bmatrix}$ a **cointegrating vector** for the state.

When applied to the nonstationary vector process $\begin{bmatrix}b_t & c_t\end{bmatrix}'$, it yields a process that is asymptotically stationary.

Equation (19) can be rearranged to take the form

$$(1-\beta)b_t + c_t = (1-\beta)\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \tag{20}$$

Equation (20) asserts that the *cointegrating residual* on the left side equals the conditional expectation of the geometric sum of future incomes on the right Section **??**.

53.4.3 Cross-Sectional Implications

Consider again (18), this time in light of our discussion of distribution dynamics in the lecture on linear systems.

The dynamics of c_t are given by

$$c_{t+1} = c_t + (1-\beta)U(I-\beta A)^{-1}Cw_{t+1} \tag{21}$$

or

$$c_t = c_0 + \sum_{j=1}^{t} \widehat{w}_j \quad \text{for} \quad \widehat{w}_{t+1} := (1-\beta)U(I-\beta A)^{-1}Cw_{t+1}$$

The unit root affecting c_t causes the time t variance of c_t to grow linearly with t.

In particular, since $\{\widehat{w}_t\}$ is IID, we have

$$\mathrm{Var}[c_t] = \mathrm{Var}[c_0] + t\,\widehat{\sigma}^2 \tag{22}$$

where

$$\widehat{\sigma}^2 := (1-\beta)^2 U(I-\beta A)^{-1}CC'(I-\beta A')^{-1}U'$$

When $\widehat{\sigma} > 0$, $\{c_t\}$ has no asymptotic distribution.

Let's consider what this means for a cross-section of ex-ante identical consumers born at time 0.

Let the distribution of c_0 represent the cross-section of initial consumption values.

Equation (22) tells us that the variance of c_t increases over time at a rate proportional to t.

A number of different studies have investigated this prediction and found some support for it (see, e.g., [42], [152]).

53.4.4 Impulse Response Functions

Impulse response functions measure responses to various impulses (i.e., temporary shocks).

The impulse response function of $\{c_t\}$ to the innovation $\{w_t\}$ is a box.

In particular, the response of c_{t+j} to a unit increase in the innovation w_{t+1} is $(1-\beta)U(I-\beta A)^{-1}C$ for all $j \geq 1$.

53.4.5 Moving Average Representation

It's useful to express the innovation to the expected present value of the endowment process in terms of a moving average representation for income y_t.

The endowment process defined by (3) has the moving average representation

$$y_{t+1} = d(L)w_{t+1} \tag{23}$$

where

- $d(L) = \sum_{j=0}^{\infty} d_j L^j$ for some sequence d_j, where L is the lag operator Section **??**
- at time t, the consumer has an information set Section **??** $w^t = [w_t, w_{t-1}, ...]$

Notice that

$$y_{t+j} - \mathbb{E}_t[y_{t+j}] = d_0 w_{t+j} + d_1 w_{t+j-1} + \cdots + d_{j-1} w_{t+1}$$

It follows that

$$\mathbb{E}_{t+1}[y_{t+j}] - \mathbb{E}_t[y_{t+j}] = d_{j-1} w_{t+1} \tag{24}$$

Using (24) in (16) gives

$$c_{t+1} - c_t = (1-\beta)d(\beta)w_{t+1} \tag{25}$$

The object $d(\beta)$ is the **present value of the moving average coefficients** in the representation for the endowment process y_t.

53.5 Two Classic Examples

We illustrate some of the preceding ideas with two examples.

In both examples, the endowment follows the process $y_t = z_{1t} + z_{2t}$ where

$$\begin{bmatrix} z_{1t+1} \\ z_{2t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} z_{1t} \\ z_{2t} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} w_{1t+1} \\ w_{2t+1} \end{bmatrix}$$

Here

- w_{t+1} is an IID 2×1 process distributed as $N(0, I)$.
- z_{1t} is a permanent component of y_t.
- z_{2t} is a purely transitory component of y_t.

53.5.1 Example 1

Assume as before that the consumer observes the state z_t at time t.

In view of (18) we have

$$c_{t+1} - c_t = \sigma_1 w_{1t+1} + (1-\beta)\sigma_2 w_{2t+1} \tag{26}$$

Formula (26) shows how an increment $\sigma_1 w_{1t+1}$ to the permanent component of income z_{1t+1} leads to

- a permanent one-for-one increase in consumption and
- no increase in savings $-b_{t+1}$

But the purely transitory component of income $\sigma_2 w_{2t+1}$ leads to a permanent increment in consumption by a fraction $1-\beta$ of transitory income.

The remaining fraction β is saved, leading to a permanent increment in $-b_{t+1}$.

Application of the formula for debt in (11) to this example shows that

$$b_{t+1} - b_t = -z_{2t} = -\sigma_2 w_{2t} \tag{27}$$

This confirms that none of $\sigma_1 w_{1t}$ is saved, while all of $\sigma_2 w_{2t}$ is saved.

The next figure illustrates these very different reactions to transitory and permanent income shocks using impulse-response functions

In [5]:
```
r = 0.05
β = 1 / (1 + r)
S = 5   # Impulse date
σ1 = σ2 = 0.15

@njit
def time_path(T, permanent=False):
    "Time path of consumption and debt given shock sequence"
    w1 = np.zeros(T+1)
    w2 = np.zeros(T+1)
    b = np.zeros(T+1)
    c = np.zeros(T+1)
    if permanent:
        w1[S+1] = 1.0
    else:
        w2[S+1] = 1.0
    for t in range(1, T):
        b[t+1] = b[t] - σ2 * w2[t]
        c[t+1] = c[t] + σ1 * w1[t+1] + (1 - β) * σ2 * w2[t+1]
    return b, c


fig, axes = plt.subplots(2, 1, figsize=(10, 8))
titles = ['transitory', 'permanent']

L = 0.175

for ax, truefalse, title in zip(axes, (True, False), titles):
    b, c = time_path(T=20, permanent=truefalse)
```

```
    ax.set_title(f'Impulse reponse: {title} income shock')
    ax.plot(c, 'g-', label="consumption")
    ax.plot(b, 'b-', label="debt")
    ax.plot((S, S), (-L, L), 'k-', lw=0.5)
    ax.grid(alpha=0.5)
    ax.set(xlabel=r'Time', ylim=(-L, L))

axes[0].legend(loc='lower right')

plt.tight_layout()
plt.show()
```

53.5.2 Example 2

Assume now that at time t the consumer observes y_t, and its history up to t, but not z_t.

Under this assumption, it is appropriate to use an *innovation representation* to form A, C, U in (18).

The discussion in sections 2.9.1 and 2.11.3 of [108] shows that the pertinent state space representation for y_t is

$$\begin{bmatrix} y_{t+1} \\ a_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & -(1-K) \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ a_t \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} a_{t+1}$$
$$y_t = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ a_t \end{bmatrix}$$

where

- $K :=$ the stationary Kalman gain
- $a_t := y_t - E[y_t \,|\, y_{t-1}, \dots, y_0]$

In the same discussion in [108] it is shown that $K \in [0, 1]$ and that K increases as σ_1/σ_2 does.

In other words, K increases as the ratio of the standard deviation of the permanent shock to that of the transitory shock increases.

Please see first look at the Kalman filter.

Applying formulas (18) implies

$$c_{t+1} - c_t = [1 - \beta(1-K)]a_{t+1} \tag{28}$$

where the endowment process can now be represented in terms of the univariate innovation to y_t as

$$y_{t+1} - y_t = a_{t+1} - (1-K)a_t \tag{29}$$

Equation (29) indicates that the consumer regards

- fraction K of an innovation a_{t+1} to y_{t+1} as *permanent*
- fraction $1-K$ as purely transitory

The consumer permanently increases his consumption by the full amount of his estimate of the permanent part of a_{t+1}, but by only $(1-\beta)$ times his estimate of the purely transitory part of a_{t+1}.

Therefore, in total, he permanently increments his consumption by a fraction $K + (1-\beta)(1-K) = 1 - \beta(1-K)$ of a_{t+1}.

He saves the remaining fraction $\beta(1-K)$.

According to equation (29), the first difference of income is a first-order moving average.

Equation (28) asserts that the first difference of consumption is IID.

Application of formula to this example shows that

$$b_{t+1} - b_t = (K-1)a_t \tag{30}$$

This indicates how the fraction K of the innovation to y_t that is regarded as permanent influences the fraction of the innovation that is saved.

53.6 Further Reading

The model described above significantly changed how economists think about consumption.

While Hall's model does a remarkably good job as a first approximation to consumption data, it's widely believed that it doesn't capture important aspects of some consumption/savings data.

For example, liquidity constraints and precautionary savings appear to be present sometimes.

Further discussion can be found in, e.g., [68], [125], [41], [31].

53.7 Appendix: The Euler Equation

Where does the first-order condition (5) come from?

Here we'll give a proof for the two-period case, which is representative of the general argument.

The finite horizon equivalent of the no-Ponzi condition is that the agent cannot end her life in debt, so $b_2 = 0$.

From the budget constraint (2) we then have

$$c_0 = \frac{b_1}{1+r} - b_0 + y_0 \quad \text{and} \quad c_1 = y_1 - b_1$$

Here b_0 and y_0 are given constants.

Substituting these constraints into our two-period objective $u(c_0) + \beta \mathbb{E}_0[u(c_1)]$ gives

$$\max_{b_1} \left\{ u\left(\frac{b_1}{R} - b_0 + y_0\right) + \beta\, \mathbb{E}_0[u(y_1 - b_1)] \right\}$$

You will be able to verify that the first-order condition is

$$u'(c_0) = \beta R\, \mathbb{E}_0[u'(c_1)]$$

Using $\beta R = 1$ gives (5) in the two-period case.

The proof for the general case is similar

Footnotes

[1] A linear marginal utility is essential for deriving (6) from (5). Suppose instead that we had imposed the following more standard assumptions on the utility function: $u'(c) > 0, u''(c) < 0, u'''(c) > 0$ and required that $c \geq 0$. The Euler equation remains (5). But the fact that $u''' < 0$ implies via Jensen's inequality that $\mathbb{E}_t[u'(c_{t+1})] > u'(\mathbb{E}_t[c_{t+1}])$. This inequality together with (5) implies that $\mathbb{E}_t[c_{t+1}] > c_t$ (consumption is said to be a 'submartingale'), so that consumption stochastically diverges to $+\infty$. The consumer's savings also diverge to $+\infty$.

[2] An optimal decision rule is a map from the current state into current actions—in this case, consumption.

[3] Representation (3) implies that $d(L) = U(I - AL)^{-1}C$.

[4] This would be the case if, for example, the spectral radius of A is strictly less than one.

[5] A moving average representation for a process y_t is said to be **fundamental** if the linear space spanned by y^t is equal to the linear space spanned by w^t. A time-invariant innovations

representation, attained via the Kalman filter, is by construction fundamental.

[6] See [90], [105], [106] for interesting applications of related ideas.

Chapter 54

Optimal Savings II: LQ Techniques

54.1 Contents

- Overview 54.2
- Setup 54.3
- The LQ Approach 54.4
- Implementation 54.5
- Two Example Economies 54.6

Co-author: Chase Coleman

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

```
In [1]: !pip install --upgrade quantecon
```

54.2 Overview

This lecture continues our analysis of the linear-quadratic (LQ) permanent income model of savings and consumption.

As we saw in our previous lecture on this topic, Robert Hall [67] used the LQ permanent income model to restrict and interpret intertemporal comovements of nondurable consumption, nonfinancial income, and financial wealth.

For example, we saw how the model asserts that for any covariance stationary process for nonfinancial income

- consumption is a random walk
- financial wealth has a unit root and is cointegrated with consumption

Other applications use the same LQ framework.

For example, a model isomorphic to the LQ permanent income model has been used by Robert Barro [14] to interpret intertemporal comovements of a government's tax collections, its expenditures net of debt service, and its public debt.

This isomorphism means that in analyzing the LQ permanent income model, we are in effect also analyzing the Barro tax smoothing model.

It is just a matter of appropriately relabeling the variables in Hall's model.

In this lecture, we'll

- show how the solution to the LQ permanent income model can be obtained using LQ control methods.
- represent the model as a linear state space system as in this lecture.
- apply QuantEcon's LinearStateSpace class to characterize statistical features of the consumer's optimal consumption and borrowing plans.

We'll then use these characterizations to construct a simple model of cross-section wealth and consumption dynamics in the spirit of Truman Bewley [22].

(Later we'll study other Bewley models—see this lecture)

The model will prove useful for illustrating concepts such as

- stationarity
- ergodicity
- ensemble moments and cross-section observations

Let's start with some imports:

In [2]:
```
import quantecon as qe
import numpy as np
import scipy.linalg as la
import matplotlib.pyplot as plt
%matplotlib inline
```

54.3 Setup

Let's recall the basic features of the model discussed in the permanent income model.

Consumer preferences are ordered by

$$E_0 \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{1}$$

where $u(c) = -(c - \gamma)^2$.

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

$$c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t, \quad t \geq 0 \tag{2}$$

and the no-Ponzi condition

$$E_0 \sum_{t=0}^{\infty} \beta^t b_t^2 < \infty \tag{3}$$

The interpretation of all variables and parameters are the same as in the previous lecture.

We continue to assume that $(1 + r)\beta = 1$.

The dynamics of $\{y_t\}$ again follow the linear state space model

$$
\begin{aligned}
z_{t+1} &= Az_t + Cw_{t+1} \\
y_t &= Uz_t
\end{aligned}
\tag{4}
$$

The restrictions on the shock process and parameters are the same as in our previous lecture.

54.3.1 Digression on a Useful Isomorphism

The LQ permanent income model of consumption is mathematically isomorphic with a version of Barro's [14] model of tax smoothing.

In the LQ permanent income model

- the household faces an exogenous process of nonfinancial income
- the household wants to smooth consumption across states and time

In the Barro tax smoothing model

- a government faces an exogenous sequence of government purchases (net of interest payments on its debt)
- a government wants to smooth tax collections across states and time

If we set

- T_t, total tax collections in Barro's model to consumption c_t in the LQ permanent income model.
- G_t, exogenous government expenditures in Barro's model to nonfinancial income y_t in the permanent income model.
- B_t, government risk-free one-period assets falling due in Barro's model to risk-free one-period consumer debt b_t falling due in the LQ permanent income model.
- R, the gross rate of return on risk-free one-period government debt in Barro's model to the gross rate of return $1 + r$ on financial assets in the permanent income model of consumption.

then the two models are mathematically equivalent.

All characterizations of a $\{c_t, y_t, b_t\}$ in the LQ permanent income model automatically apply to a $\{T_t, G_t, B_t\}$ process in the Barro model of tax smoothing.

See consumption and tax smoothing models for further exploitation of an isomorphism between consumption and tax smoothing models.

54.3.2 A Specification of the Nonfinancial Income Process

For the purposes of this lecture, let's assume $\{y_t\}$ is a second-order univariate autoregressive process:

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

We can map this into the linear state space framework in (4), as discussed in our lecture on linear models.

To do so we take

$$z_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ \sigma \\ 0 \end{bmatrix}, \quad \text{and} \quad U = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

54.4 The LQ Approach

Previously we solved the permanent income model by solving a system of linear expectational difference equations subject to two boundary conditions.

Here we solve the same model using LQ methods based on dynamic programming.

After confirming that answers produced by the two methods agree, we apply QuantEcon's LinearStateSpace class to illustrate features of the model.

Why solve a model in two distinct ways?

Because by doing so we gather insights about the structure of the model.

Our earlier approach based on solving a system of expectational difference equations brought to the fore the role of the consumer's expectations about future nonfinancial income.

On the other hand, formulating the model in terms of an LQ dynamic programming problem reminds us that

- finding the state (of a dynamic programming problem) is an art, and
- iterations on a Bellman equation implicitly jointly solve both a forecasting problem and a control problem

54.4.1 The LQ Problem

Recall from our lecture on LQ theory that the optimal linear regulator problem is to choose a decision rule for u_t to minimize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t \{x_t' R x_t + u_t' Q u_t\},$$

subject to x_0 given and the law of motion

$$x_{t+1} = \tilde{A} x_t + \tilde{B} u_t + \tilde{C} w_{t+1}, \qquad t \geq 0, \tag{5}$$

where w_{t+1} is IID with mean vector zero and $\mathbb{E} w_t w_t' = I$.

The tildes in $\tilde{A}, \tilde{B}, \tilde{C}$ are to avoid clashing with notation in (4).

The value function for this problem is $v(x) = -x' P x - d$, where

- P is the unique positive semidefinite solution of the corresponding matrix Riccati equation.
- The scalar d is given by $d = \beta(1-\beta)^{-1} \text{trace}(P \tilde{C} \tilde{C}')$.

The optimal policy is $u_t = -F x_t$, where $F := \beta(Q + \beta \tilde{B}' P \tilde{B})^{-1} \tilde{B}' P \tilde{A}$.

Under an optimal decision rule F, the state vector x_t evolves according to $x_{t+1} = (\tilde{A} - \tilde{B} F) x_t + \tilde{C} w_{t+1}$.

54.4.2 Mapping into the LQ Framework

To map into the LQ framework, we'll use

$$x_t := \begin{bmatrix} z_t \\ b_t \end{bmatrix} = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \\ b_t \end{bmatrix}$$

as the state vector and $u_t := c_t - \gamma$ as the control.

With this notation and $U_\gamma := \begin{bmatrix} \gamma & 0 & 0 \end{bmatrix}$, we can write the state dynamics as in (5) when

$$\tilde{A} := \begin{bmatrix} A & 0 \\ (1+r)(U_\gamma - U) & 1+r \end{bmatrix} \quad \tilde{B} := \begin{bmatrix} 0 \\ 1+r \end{bmatrix} \quad \text{and} \quad \tilde{C} := \begin{bmatrix} C \\ 0 \end{bmatrix} w_{t+1}$$

Please confirm for yourself that, with these definitions, the LQ dynamics (5) match the dynamics of z_t and b_t described above.

To map utility into the quadratic form $x_t' R x_t + u_t' Q u_t$ we can set

- $Q := 1$ (remember that we are minimizing) and
- $R :=$ a 4×4 matrix of zeros

However, there is one problem remaining.

We have no direct way to capture the non-recursive restriction (3) on the debt sequence $\{b_t\}$ from within the LQ framework.

To try to enforce it, we're going to use a trick: put a small penalty on b_t^2 in the criterion function.

In the present setting, this means adding a small entry $\epsilon > 0$ in the $(4, 4)$ position of R.

That will induce a (hopefully) small approximation error in the decision rule.

We'll check whether it really is small numerically soon.

54.5 Implementation

Let's write some code to solve the model.

One comment before we start is that the bliss level of consumption γ in the utility function has no effect on the optimal decision rule.

We saw this in the previous lecture permanent income.

The reason is that it drops out of the Euler equation for consumption.

In what follows we set it equal to unity.

54.5.1 The Exogenous Nonfinancial Income Process

First, we create the objects for the optimal linear regulator

```
In [3]: # Set parameters
        α, β, ρ1, ρ2, σ = 10.0, 0.95, 0.9, 0.0, 1.0
```

```
R = 1 / β
A = np.array([[1., 0., 0.],
              [α,  ρ1, ρ2],
              [0., 1., 0.]])
C = np.array([[0.], [σ], [0.]])
G = np.array([[0., 1., 0.]])

# Form LinearStateSpace system and pull off steady state moments
μ_z0 = np.array([[1.0], [0.0], [0.0]])
Σ_z0 = np.zeros((3, 3))
Lz = qe.LinearStateSpace(A, C, G, mu_0=μ_z0, Sigma_0=Σ_z0)
μ_z, μ_y, Σ_z, Σ_y = Lz.stationary_distributions()

# Mean vector of state for the savings problem
mxo = np.vstack([μ_z, 0.0])

# Create stationary covariance matrix of x -- start everyone off at b=0
a1 = np.zeros((3, 1))
aa = np.hstack([Σ_z, a1])
bb = np.zeros((1, 4))
sxo = np.vstack([aa, bb])

# These choices will initialize the state vector of an individual at zero
# debt and the ergodic distribution of the endowment process. Use these to
# create the Bewley economy.
mxbewley = mxo
sxbewley = sxo
```

The next step is to create the matrices for the LQ system

```
In [4]: A12 = np.zeros((3,1))
        ALQ_l = np.hstack([A, A12])
        ALQ_r = np.array([[0, -R, 0, R]])
        ALQ = np.vstack([ALQ_l, ALQ_r])

        RLQ = np.array([[0., 0., 0., 0.],
                        [0., 0., 0., 0.],
                        [0., 0., 0., 0.],
                        [0., 0., 0., 1e-9]])

        QLQ = np.array([1.0])
        BLQ = np.array([0., 0., 0., R]).reshape(4,1)
        CLQ = np.array([0., σ, 0., 0.]).reshape(4,1)
        β_LQ = β
```

Let's print these out and have a look at them

```
In [5]: print(f"A = \n {ALQ}")
        print(f"B = \n {BLQ}")
        print(f"R = \n {RLQ}")
        print(f"Q = \n {QLQ}")
```

```
   A =
[[ 1.          0.          0.          0.        ]
[10.          0.9         0.          0.        ]
```

```
 [ 0.          1.          0.          0.        ]
 [ 0.         -1.05263158  0.          1.05263158]]
B =
 [[0.        ]
 [0.        ]
 [0.        ]
 [1.05263158]]
R =
 [[0.e+00 0.e+00 0.e+00 0.e+00]
 [0.e+00 0.e+00 0.e+00 0.e+00]
 [0.e+00 0.e+00 0.e+00 0.e+00]
 [0.e+00 0.e+00 0.e+00 1.e-09]]
Q =
 [1.]
```

Now create the appropriate instance of an LQ model

```
In [6]: lqpi = qe.LQ(QLQ, RLQ, ALQ, BLQ, C=CLQ, beta=β_LQ)
```

We'll save the implied optimal policy function soon compare them with what we get by employing an alternative solution method

```
In [7]: P, F, d = lqpi.stationary_values()  # Compute value function and decision
 ↪rule
        ABF = ALQ - BLQ @ F  #  Form closed loop system
```

54.5.2 Comparison with the Difference Equation Approach

In our first lecture on the infinite horizon permanent income problem we used a different solution method.

The method was based around

- deducing the Euler equations that are the first-order conditions with respect to consumption and savings.
- using the budget constraints and boundary condition to complete a system of expectational linear difference equations.
- solving those equations to obtain the solution.

Expressed in state space notation, the solution took the form

$$
\begin{aligned}
z_{t+1} &= Az_t + Cw_{t+1} \\
b_{t+1} &= b_t + U[(I - \beta A)^{-1}(A - I)]z_t \\
y_t &= Uz_t \\
c_t &= (1-\beta)[U(I-\beta A)^{-1}z_t - b_t]
\end{aligned}
$$

Now we'll apply the formulas in this system

```
In [8]: # Use the above formulas to create the optimal policies for b_{t+1} and c_t
        b_pol = G @ la.inv(np.eye(3, 3) - β * A) @ (A - np.eye(3, 3))
        c_pol = (1 - β) * G @ la.inv(np.eye(3, 3) - β * A)

        # Create the A matrix for a LinearStateSpace instance
```

```
A_LSS1 = np.vstack([A, b_pol])
A_LSS2 = np.eye(4, 1, -3)
A_LSS = np.hstack([A_LSS1, A_LSS2])

# Create the C matrix for LSS methods
C_LSS = np.vstack([C, np.zeros(1)])

# Create the G matrix for LSS methods
G_LSS1 = np.vstack([G, c_pol])
G_LSS2 = np.vstack([np.zeros(1), -(1 - β)])
G_LSS = np.hstack([G_LSS1, G_LSS2])

# Use the following values to start everyone off at b=0, initial incomes zero
μ_0 = np.array([1., 0., 0., 0.])
Σ_0 = np.zeros((4, 4))
```

A_LSS calculated as we have here should equal ABF calculated above using the LQ model

```
In [9]: ABF - A_LSS
```

```
Out[9]: array([[ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
                 0.00000000e+00],
               [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
                 0.00000000e+00],
               [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
                 0.00000000e+00],
               [-9.51248178e-06,  9.51247915e-08,  3.36117263e-17,
                -1.99999923e-08]])
```

Now compare pertinent elements of c_pol and F

```
In [10]: print(c_pol, "\n", -F)
```

```
   [[6.55172414e+01 3.44827586e-01 1.68058632e-18]]
 [[ 6.55172323e+01  3.44827677e-01 -0.00000000e+00 -5.00000190e-02]]
```

We have verified that the two methods give the same solution.

Now let's create instances of the LinearStateSpace class and use it to do some interesting experiments.

To do this, we'll use the outcomes from our second method.

54.6 Two Example Economies

In the spirit of Bewley models [22], we'll generate panels of consumers.

The examples differ only in the initial states with which we endow the consumers.

All other parameter values are kept the same in the two examples

- In the first example, all consumers begin with zero nonfinancial income and zero debt.
  - The consumers are thus *ex-ante* identical.

- In the second example, while all begin with zero debt, we draw their initial income levels from the invariant distribution of financial income.
  - Consumers are *ex-ante* heterogeneous.

In the first example, consumers' nonfinancial income paths display pronounced transients early in the sample

- these will affect outcomes in striking ways

Those transient effects will not be present in the second example.

We use methods affiliated with the LinearStateSpace class to simulate the model.

54.6.1 First Set of Initial Conditions

We generate 25 paths of the exogenous non-financial income process and the associated optimal consumption and debt paths.

In the first set of graphs, darker lines depict a particular sample path, while the lighter lines describe 24 other paths.

A second graph plots a collection of simulations against the population distribution that we extract from the `LinearStateSpace` instance `LSS`.

Comparing sample paths with population distributions at each date t is a useful exercise—see our discussion of the laws of large numbers

```
In [11]: lss = qe.LinearStateSpace(A_LSS, C_LSS, G_LSS, mu_0=μ_0, Sigma_0=Σ_0)
```

54.6.2 Population and Sample Panels

In the code below, we use the LinearStateSpace class to

- compute and plot population quantiles of the distributions of consumption and debt for a population of consumers.
- simulate a group of 25 consumers and plot sample paths on the same graph as the population distribution.

```
In [12]: def income_consumption_debt_series(A, C, G, μ_0, Σ_0, T=150, npaths=25):
             """
             This function takes initial conditions (μ_0, Σ_0) and uses the
             LinearStateSpace class from QuantEcon to  simulate an economy
             npaths times for T periods. It then uses that information to
             generate some graphs related to the discussion below.
             """
             lss = qe.LinearStateSpace(A, C, G, mu_0=μ_0, Sigma_0=Σ_0)

             # Simulation/Moment Parameters
             moment_generator = lss.moment_sequence()

             # Simulate various paths
             bsim = np.empty((npaths, T))
             csim = np.empty((npaths, T))
             ysim = np.empty((npaths, T))

             for i in range(npaths):
```

```
        sims = lss.simulate(T)
        bsim[i, :] = sims[0][-1, :]
        csim[i, :] = sims[1][1, :]
        ysim[i, :] = sims[1][0, :]

    # Get the moments
    cons_mean = np.empty(T)
    cons_var = np.empty(T)
    debt_mean = np.empty(T)
    debt_var = np.empty(T)
    for t in range(T):
        μ_x, μ_y, Σ_x, Σ_y = next(moment_generator)
        cons_mean[t], cons_var[t] = μ_y[1], Σ_y[1, 1]
        debt_mean[t], debt_var[t] = μ_x[3], Σ_x[3, 3]

    return bsim, csim, ysim, cons_mean, cons_var, debt_mean, debt_var

def consumption_income_debt_figure(bsim, csim, ysim):

    # Get T
    T =  bsim.shape[1]

    # Create the first figure
    fig, ax = plt.subplots(2, 1, figsize=(10, 8))
    xvals = np.arange(T)

    # Plot consumption and income
    ax[0].plot(csim[0, :], label="c", color="b")
    ax[0].plot(ysim[0, :], label="y", color="g")
    ax[0].plot(csim.T, alpha=.1, color="b")
    ax[0].plot(ysim.T, alpha=.1, color="g")
    ax[0].legend(loc=4)
    ax[0].set(title="Nonfinancial Income, Consumption, and Debt",
              xlabel="t", ylabel="y and c")

    # Plot debt
    ax[1].plot(bsim[0, :], label="b", color="r")
    ax[1].plot(bsim.T, alpha=.1, color="r")
    ax[1].legend(loc=4)
    ax[1].set(xlabel="t", ylabel="debt")

    fig.tight_layout()
    return fig

def consumption_debt_fanchart(csim, cons_mean, cons_var,
                              bsim, debt_mean, debt_var):
    # Get T
    T =  bsim.shape[1]

    # Create percentiles of cross-section distributions
    cmean = np.mean(cons_mean)
    c90 = 1.65 * np.sqrt(cons_var)
    c95 = 1.96 * np.sqrt(cons_var)
    c_perc_95p, c_perc_95m = cons_mean + c95, cons_mean - c95
    c_perc_90p, c_perc_90m = cons_mean + c90, cons_mean - c90

    # Create percentiles of cross-section distributions
    dmean = np.mean(debt_mean)
```

```
    d90 = 1.65 * np.sqrt(debt_var)
    d95 = 1.96 * np.sqrt(debt_var)
    d_perc_95p, d_perc_95m = debt_mean + d95, debt_mean - d95
    d_perc_90p, d_perc_90m = debt_mean + d90, debt_mean - d90


    # Create second figure
    fig, ax = plt.subplots(2, 1, figsize=(10, 8))
    xvals = np.arange(T)

    # Consumption fan
    ax[0].plot(xvals, cons_mean, color="k")
    ax[0].plot(csim.T, color="k", alpha=.25)
   ax[0].fill_between(xvals, c_perc_95m, c_perc_95p, alpha=.25, color="b")
   ax[0].fill_between(xvals, c_perc_90m, c_perc_90p, alpha=.25, color="r")
    ax[0].set(title="Consumption/Debt over time",
              ylim=(cmean-15, cmean+15), ylabel="consumption")

    # Debt fan
    ax[1].plot(xvals, debt_mean, color="k")
    ax[1].plot(bsim.T, color="k", alpha=.25)
   ax[1].fill_between(xvals, d_perc_95m, d_perc_95p, alpha=.25, color="b")
   ax[1].fill_between(xvals, d_perc_90m, d_perc_90p, alpha=.25, color="r")
    ax[1].set(xlabel="t", ylabel="debt")

    fig.tight_layout()
    return fig
```

Now let's create figures with initial conditions of zero for y_0 and b_0

```
In [13]: out = income_consumption_debt_series(A_LSS, C_LSS, G_LSS, μ_0, Σ_0)
         bsim0, csim0, ysim0 = out[:3]
         cons_mean0, cons_var0, debt_mean0, debt_var0 = out[3:]

         consumption_income_debt_figure(bsim0, csim0, ysim0)

         plt.show()
```

```
In [14]: consumption_debt_fanchart(csim0, cons_mean0, cons_var0,
                                   bsim0, debt_mean0, debt_var0)

         plt.show()
```

Here is what is going on in the above graphs.

For our simulation, we have set initial conditions $b_0 = y_{-1} = y_{-2} = 0$.

Because $y_{-1} = y_{-2} = 0$, nonfinancial income y_t starts far below its stationary mean $\mu_{y,\infty}$ and rises early in each simulation.

Recall from the previous lecture that we can represent the optimal decision rule for consumption in terms of the **co-integrating relationship**

$$(1-\beta)b_t + c_t = (1-\beta)E_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \tag{6}$$

So at time 0 we have

$$c_0 = (1-\beta)E_0 \sum_{t=0}^{\infty} \beta^j y_t$$

This tells us that consumption starts at the income that would be paid by an annuity whose value equals the expected discounted value of nonfinancial income at time $t = 0$.

To support that level of consumption, the consumer borrows a lot early and consequently builds up substantial debt.

In fact, he or she incurs so much debt that eventually, in the stochastic steady state, he consumes less each period than his nonfinancial income.

He uses the gap between consumption and nonfinancial income mostly to service the interest payments due on his debt.

Thus, when we look at the panel of debt in the accompanying graph, we see that this is a group of *ex-ante* identical people each of whom starts with zero debt.

All of them accumulate debt in anticipation of rising nonfinancial income.

They expect their nonfinancial income to rise toward the invariant distribution of income, a consequence of our having started them at $y_{-1} = y_{-2} = 0$.

Cointegration Residual

The following figure plots realizations of the left side of (6), which, as discussed in our last lecture, is called the **cointegrating residual**.

As mentioned above, the right side can be thought of as an annuity payment on the expected present value of future income $E_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$.

Early along a realization, c_t is approximately constant while $(1 - \beta)b_t$ and $(1 - \beta)E_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$ both rise markedly as the household's present value of income and borrowing rise pretty much together.

This example illustrates the following point: the definition of cointegration implies that the cointegrating residual is *asymptotically* covariance stationary, not *covariance stationary*.

The cointegrating residual for the specification with zero income and zero debt initially has a notable transient component that dominates its behavior early in the sample.

By altering initial conditions, we shall remove this transient in our second example to be presented below

```
In [15]: def cointegration_figure(bsim, csim):
             """
             Plots the cointegration
             """
             # Create figure
             fig, ax = plt.subplots(figsize=(10, 8))
             ax.plot((1 - β) * bsim[0, :] + csim[0, :], color="k")
             ax.plot((1 - β) * bsim.T + csim.T, color="k", alpha=.1)

             ax.set(title="Cointegration of Assets and Consumption", xlabel="t")

             return fig
```

```
In [16]: cointegration_figure(bsim0, csim0)
         plt.show()
```

54.6.3 A "Borrowers and Lenders" Closed Economy

When we set $y_{-1} = y_{-2} = 0$ and $b_0 = 0$ in the preceding exercise, we make debt "head north" early in the sample.

Average debt in the cross-section rises and approaches the asymptote.

We can regard these as outcomes of a "small open economy" that borrows from abroad at the fixed gross interest rate $R = r + 1$ in anticipation of rising incomes.

So with the economic primitives set as above, the economy converges to a steady state in which there is an excess aggregate supply of risk-free loans at a gross interest rate of R.

This excess supply is filled by "foreigner lenders" willing to make those loans.

We can use virtually the same code to rig a "poor man's Bewley [22] model" in the following way

- as before, we start everyone at $b_0 = 0$.
- But instead of starting everyone at $y_{-1} = y_{-2} = 0$, we draw $\begin{bmatrix} y_{-1} \\ y_{-2} \end{bmatrix}$ from the invariant distribution of the $\{y_t\}$ process.

This rigs a closed economy in which people are borrowing and lending with each other at a gross risk-free interest rate of $R = \beta^{-1}$.

Across the group of people being analyzed, risk-free loans are in zero excess supply.

We have arranged primitives so that $R = \beta^{-1}$ clears the market for risk-free loans at zero aggregate excess supply.

So the risk-free loans are being made from one person to another within our closed set of agent.

There is no need for foreigners to lend to our group.

Let's have a look at the corresponding figures

```
In [17]: out = income_consumption_debt_series(A_LSS, C_LSS, G_LSS, mxbewley,
         sxbewley)
         bsimb, csimb, ysimb = out[:3]
         cons_meanb, cons_varb, debt_meanb, debt_varb = out[3:]

         consumption_income_debt_figure(bsimb, csimb, ysimb)

         plt.show()
```

```
In [18]: consumption_debt_fanchart(csimb, cons_meanb, cons_varb,
                                   bsimb, debt_meanb, debt_varb)

         plt.show()
```

The graphs confirm the following outcomes:

- As before, the consumption distribution spreads out over time.

But now there is some initial dispersion because there is *ex-ante* heterogeneity in the initial draws of $\begin{bmatrix} y_{-1} \\ y_{-2} \end{bmatrix}$.

- As before, the cross-section distribution of debt spreads out over time.
- Unlike before, the average level of debt stays at zero, confirming that this is a closed borrower-and-lender economy.
- Now the cointegrating residual seems stationary, and not just asymptotically stationary.

Let's have a look at the cointegration figure

```
In [19]: cointegration_figure(bsimb, csimb)
         plt.show()
```

Cointegration of Assets and Consumption
102
101
100
99
98
97
0
20
40
60
80
100
120
140
t

Chapter 55

Information and Consumption Smoothing

55.1 Contents

- Overview 55.2
- Two Representations of the **Same** Nonfinancial Income Process **??**
- State Space Representations 55.4

Co-author: Zejin Shi

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

```
In [1]: !pip install --upgrade quantecon
```

55.2 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 [**?**].

55.3 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

$$y_{t+1} - y_t = \epsilon_{t+1} - \beta^{-1}\epsilon_t \tag{1}$$

where $\{\epsilon_t\}$ is an i.i.d. normally distributed scalar process with means of zero and contemporaneous variances σ_ϵ^2.

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

Furthermore, as we'll see below, representation (1) has the peculiar property that a positive shock ϵ_{t+1} leaves the discounted present value of the consumer's financial income at time $t+1$ unaltered.

The second representation of the **same** $\{y_t\}$ process is

$$y_{t+1} - y_t = a_{t+1} - \beta a_t \tag{2}$$

where $\{a_t\}$ is another i.i.d. normally distributed scalar process, with means of zero and now variances σ_a^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 β^{-2}.

The second representation is the **innovations representation** from Kalman filtering theory.

To see how this works, note that equating representations (1) and (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,

$$a_{t+1} = \epsilon_{t+1} + (\beta - \beta^{-1}) \sum_{j=0}^{\infty} \beta^j \epsilon_{t-j} \tag{3}$$

which we can also write as

$$a_{t+1} = \sum_{j=0}^{\infty} \epsilon_{t+1-j} \equiv h(L)\epsilon_{t+1}$$

where L is the one-period lag operator, I is the identity operator, and

$$h(L) = \frac{I - \beta^{-1}L}{I - \beta L}$$

Let $c_j \equiv E z_t z_{t-j}$ be the jth autocovariance of the $\{y_t - y_{t-1}\}$ process.

Using calculations in the quantecon lecture, where $z \in C$ is a complex variable, the covariance generating function $g(z) = \sum_{j=-\infty}^{\infty} c_j z^j$ 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 σ_a^2 from formula (3) according to

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

55.3.1 Application of Kalman filter

We can also obtain representation (2) from representation (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

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

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.

55.3.2 News Shocks and Less Informative Shocks

Representation (1) is cast in terms of a **news shock** ϵ_{t+1} that represents a shock to nonfinancial income coming from taxes, transfers, and other random sources of income changes known to a well-informed person having all sorts of information about the income process.

Representation (2) for the **same** income process is driven by shocks a_t that contain less information than the news shock ϵ_t.

Representation (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 $y_t, y_{t-1}, ...$

The shock ϵ_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 (3) reveals the important fact that the **original shock** ϵ_t contains more information about future y's than is contained in the semi-infinite history $y^t = [y_t, y_{t-1}, ...]$ of current and past y's.

Staring at representation (3) for a_{t+1} shows that it consists both of **new news** ϵ_{t+1} as well as a long moving average $(\beta - \beta^{-1})\sum_{j=0}^{\infty}\beta^j\epsilon_{t-j}$ of **old news**.

The **better informed** representation (1) asserts that a shock ϵ_t results in an impulse response to nonfinancial income of ϵ_t times the sequence

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

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

Because $1-\beta^{-1} < 0$, this means that a positive shock of ϵ_t today raises income at time t by ϵ_t and then **decreases all** future incomes by $(\beta^{-1}-1)\epsilon_t$.

This pattern precisely describes the following mental experiment:

- The consumer receives a government transfer of ϵ_t at time t.
- The government finances the transfer by issuing a one-period bond on which it pays a gross one-period risk-free interest rate equal to β^{-1}.
- In each future period, the government **rolls over** the one-period bond and so continues to borrow ϵ_t forever.
- The government imposes a lump-sum tax on the consumer in order to pay just the current interest on the original bond and its successors created by the roll-over operation.
- In all future periods $t+1, t+2, ...$, the government levies a lump-sum tax on the consumer of $\beta^{-1}-1$ that is just enough to pay the interest on the bond.

The **present value** of the impulse response or moving average coefficients equals $d_\epsilon(L) = \frac{0}{1-\beta} = 0$, a fact that we'll see again below.

Representation (2), i.e., the innovation representation, asserts that a shock a_t results in an impulse response to nonfinancial income of a_t times

$$1, 1-\beta, 1-\beta, ...$$

so that a shock that increases income y_t by a_t at time t can be expected to be followed by an **increase** in y_{t+j} of a_t times $1-\beta > 0$ in all future periods $j = 1, 2, ...$

The present value of the impulse response or moving average coefficients for representation (2) is $d_a(\beta) = \frac{1-\beta^2}{1-\beta} = (1+\beta)$, another fact that will be important below.

55.3.3 Representation of ϵ_t in Terms of Future y's

Notice that reprentation (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 ϵ_t equals β 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, ϵ_t contains **exact** information about an important linear combination of **future** nonfinancial income.

55.3.4 Representation in Terms of a_t Shocks

Next notice that representation (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

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

55.3.5 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 (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 (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 (1) responds to each shock ϵ_{t+1} by leaving his consumption unaltered and **saving** all of a_{t+1} in anticipation of the permanently increased taxes that he will bear to pay for the addition a_{t+1} to his time $t+1$ nonfinancial income.

The consumer with information associated with representation (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.

55.4 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.

55.4.1 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 β is the **same** value as the discount factor in the household's problem in the LQ savings problem in the lecture.

For these two representations, we use the code in 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 ϵ_t and a_t for the two representations.
- run simulations of $\{y_t, c_t, b_t\}$ of length T under both of the representations (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 γ has no effect on the impulse response functions that interest us.)

The state transition equations under our two representations for the nonfinancial income process $\{y_t\}$ can be written as

$$\begin{bmatrix} y_{t+1} \\ \epsilon_{t+1} \\ b_{t+1} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & -\beta^{-1} & 0 \\ 0 & 0 & 0 \\ -(1+r) & 0 & 1+r \end{bmatrix}}_{\equiv A_1} \begin{bmatrix} y_t \\ \epsilon_t \\ b_t \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 1+r \end{bmatrix}}_{\equiv B_1} \begin{bmatrix} c_t \end{bmatrix} + \underbrace{\begin{bmatrix} \sigma_\epsilon \\ \sigma_\epsilon \\ 0 \end{bmatrix}}_{\equiv C_1} \nu_{t+1},$$

and

$$\begin{bmatrix} y_{t+1} \\ a_{t+1} \\ b_{t+1} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & -\beta & 0 \\ 0 & 0 & 0 \\ -(1+r) & 0 & 1+r \end{bmatrix}}_{\equiv A_2} \begin{bmatrix} y_t \\ a_t \\ b_t \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 1+r \end{bmatrix}}_{\equiv B_2} \begin{bmatrix} c_t \end{bmatrix} + \underbrace{\begin{bmatrix} \sigma_a \\ \sigma_a \\ 0 \end{bmatrix}}_{\equiv C_2} u_{t+1}.$$

As usual, we start by importing packages.

In [2]:
```
import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
%matplotlib inline
```

In [3]:
```
# 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.])
```

In [4]:
```
# Original representation state transition matrices
ALQ1 = np.array([[1, -R, 0],
                 [0, 0, 0],
                 [-R, 0, R]])
BLQ1 = np.array([[0, 0, R]]).T
CLQ1 = np.array([[σϵ, σϵ, 0]]).T

# Construct and solve the LQ problem
LQ1 = qe.LQ(QLQ, RLQ, ALQ1, BLQ1, C=CLQ1, beta=β)
P1, F1, d1 = LQ1.stationary_values()
```

In [5]:
```
# The optimal decision rule for c
-F1
```

Out[5]:
```
array([[ 1.  , -1.  , -0.05]])
```

Evidently optimal consumption and debt decision rules for the consumer having news representation (1) are

$$
\begin{aligned}
c_t^* &= y_t - \epsilon_t - (1-\beta)\, b_t, \\
b_{t+1}^* &= \beta^{-1} c_t^* + \beta^{-1} b_t - \beta^{-1} y_t \\
&= \beta^{-1} y_t - \beta^{-1}\epsilon_t - (\beta^{-1} - 1)\, b_t + \beta^{-1} b_t - \beta^{-1} y_t \\
&= b_t - \beta^{-1}\epsilon_t.
\end{aligned}
$$

In [6]:
```
# 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()
```

In [7]:
```
-F2
```

Out[7]:
```
array([[ 1.    , -0.9025, -0.05  ]])
```

For a consumer having access only to the information associated with the innovations representation (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.

In [8]:
```
# 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.

In [9]:
```
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)])
```

In [10]:
```
c_res1 / σϵ, b_res1 / σϵ
```

Out[10]:
```
(array([1.99997796e-11, 1.89473992e-11, 1.78947621e-11, 1.68421319e-11,
        1.57894947e-11]),
 array([ 0.        , -1.05263158, -1.05263158, -1.05263158, -1.05263158]))
```

In [11]:
```
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()
```

Out[11]:
```
<matplotlib.legend.Legend at 0x7f4aacb735f8>
```

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 ϵ_t is to leave his consumption unchanged and to save the entire amount of his extra income and then forever roll over the extra bonds that he holds.

To see this notice, that starting from next period on, his debt permanently **decreases** by β^{-1}

```
In [12]: c_res2 / σa, b_res2 / σa
```

```
Out[12]: (array([0.0975, 0.0975, 0.0975, 0.0975, 0.0975]),
          array([ 0.  , -0.95, -0.95, -0.95, -0.95]))
```

```
In [13]: 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()
```

```
Out[13]: <matplotlib.legend.Legend at 0x7f4aacaa4828>
```

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.

In [14]:
```
# Set time length for simulation
T = 100
```

In [15]:
```
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()
```

Out[15]:
```
<matplotlib.legend.Legend at 0x7f4aac654f98>
```

```
In [16]: 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()

Out[16]: <matplotlib.legend.Legend at 0x7f4aac5c2f60>
```

55.4.2 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

$$y_{t+1} - y_t = -\beta^{-1}\epsilon_t + \epsilon_{t+1}.$$

1. We take the **same** $\{y_t\}$ realization generated in step 1 and form an innovation process $\{a_t\}$ from the formulas

$$\begin{aligned} a_0 &= 0 \\ a_t &= \sum_{j=0}^{t-1} \beta^j (y_{t-j} - y_{t-j-1}) + \beta^t a_0, \quad t \geq 1 \end{aligned}$$

1. 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.
2. 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.

55.4.3 Calculating Innovations in Another Way

Here we use formula (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

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

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

55.4.4 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 [**?**] in their analysis of **fiscal foresight**.

Chapter 56

Consumption Smoothing with Complete and Incomplete Markets

56.1 Contents

- Overview 56.2
- Background 56.3
- Linear State Space Version of Complete Markets Model 56.4
- Model 1 (Complete Markets) 56.5
- Model 2 (One-Period Risk-Free Debt Only) 56.6

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

```
In [1]: !pip install --upgrade quantecon
```

56.2 Overview

This lecture describes two types of consumption-smoothing models.

- one is in the **complete markets** tradition of Kenneth Arrow https://en.wikipedia.org/wiki/Kenneth_Arrow
- the other is in the **incomplete markets** tradition of Hall [67]

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 [67] worked in an incomplete markets tradition by assuming that the only asset that can be traded is a risk-free one period bond.

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

This is equivalent to saying that it costs β^{-1} of time t consumption to buy one unit of consumption at time $t + 1$ for sure.

So β^{-1} 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:

In [2]:
```
import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
%matplotlib inline
import scipy.linalg as la
```

56.2.1 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

56.3 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 https://en.wikipedia.org/wiki/Kenneth_Arrow

In the **incomplete markets version**, the consumer can buy and sell only one security each period, a risk-free one-period bond with gross one-period return β^{-1}.

56.4 Linear State Space Version of Complete Markets Model

Now 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, there are

- incomplete markets: the consumer can trade only a single risk-free one-period bond bearing gross one-period risk-free interest rate equal to β^{-1}.
- the consumer's exogenous nonfinancial income is governed by a linear state space model driven by Gaussian shocks, the kind of model studied in an earlier lecture about linear state space models.

Let's write down a complete markets counterpart of that model.

Suppose that nonfinancial income is governed by the state space system

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= S_y x_t \end{aligned}$$

where x_t is an $n \times 1$ vector and $w_{t+1} \sim N(0, I)$ is IID over time.

We want a natural counterpart of the Hall assumption that the one-period risk-free gross interest rate is β^{-1}.

Accordingly, we assume that prices of one-period ahead Arrow securities are described by the **pricing kernel**

$$q_{t+1}(x_{t+1} \,|\, x_t) = \beta \phi(x_{t+1} \,|\, Ax_t, CC') \tag{1}$$

where $\phi(\cdot \,|\, \mu, \Sigma)$ is a multivariate Gaussian distribution with mean vector μ and covariance matrix Σ.

With the pricing kernel $q_{t+1}(x_{t+1} \,|\, x_t)$ in hand, we can price claims to consumption at time $t+1$ consumption that pay off when $x_{t+1} \in A$ at time $t+1$:

$$\int_A q_{t+1}(x_{t+1} \,|\, x_t) dx_{t+1}$$

where A is a subset of $\mathbb{R}^n$.

The price $\int_A q_{t+1}(x_{t+1} \,|\, x_t) dx_{t+1}$ 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 $q(x_{t+1} \,|\, x_t)$.

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 (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

$$E_t b_{t+1} = \int \phi_{t+1}(x_{t+1}|x_t) b_{t+1}(x_{t+1}) dx_{t+1}$$

which verifies that $E_t b_{t+1}$ 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

$$b_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j (y_{t+j} - c_{t+j})$$

We assume as before that 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 $u'' < 0$.

The 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

$$b_t = S_y (I - \beta A)^{-1} x_t - \frac{1}{1-\beta} \bar{c} \tag{2}$$

where $\bar{c}$ satisfies

$$\bar{b}_0 = S_y(I - \beta A)^{-1} x_0 - \frac{1}{1-\beta}\bar{c} \tag{3}$$

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 (3) and the consumer's debt is the fixed function of the state x_t described by (2).

Please recall that in the LQ permanent income model studied in first lecture on the 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

In [3]:
```
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.ones(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=(15, 5))

# Consumption plots
ax[0].set_title('Consumption and income')
ax[0].plot(np.arange(N_simul), c_hist_com, label='consumption')
ax[0].plot(np.arange(N_simul), y_hist_com, label='income', alpha=.6,
↪linestyle='--')
ax[0].legend()
ax[0].set_xlabel('Periods')
ax[0].set_ylim([80, 120])

# Debt plots
ax[1].set_title('Debt and income')
ax[1].plot(np.arange(N_simul), b_hist_com, label='debt')
ax[1].plot(np.arange(N_simul), y_hist_com, label='Income', alpha=.6,
↪linestyle='--')
ax[1].legend()
ax[1].axhline(0, color='k')
ax[1].set_xlabel('Periods')

plt.show()
```

56.4.1 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.

56.4.2 Incomplete Markets Version

The incomplete markets version of the model with nonfinancial income being governed by a linear state space system is described in the first lecture on the permanent income model and the followup lecture on the 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.

56.4.3 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 models, nonfinancial income is governed by a two-state Markov chain (it's easy to generalize this to an N state Markov chain).

In particular, the *state* $s_t \in \{1, 2\}$ follows a Markov chain with transition probability matrix

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

where $\mathbb{P}$ means conditional probability

Nonfinancial income $\{y_t\}$ obeys

$$y_t = \begin{cases} \bar{y}_1 & \text{if } s_t = 1 \\ \bar{y}_2 & \text{if } s_t = 2 \end{cases}$$

A consumer wishes to maximize

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(c_t)\right] \quad \text{where} \quad u(c_t) = -(c_t - \gamma)^2 \quad \text{and} \quad 0 < \beta < 1 \tag{4}$$

Here $\gamma > 0$ is a bliss level of consumption

56.4.4 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

56.5 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

$$c_t + b_t \leq y(s_t) + \sum_j q(j \mid s_t)\, b_{t+1}(j \mid s_t) \tag{5}$$

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.

These are

- when multiplied by $q(j \mid s_t)$, a source of time t **revenues** for the consumer
- a source of time $t+1$ **expenditures** for the consumer

A natural analog of Hall's assumption that the one-period risk-free gross interest rate is β^{-1} is

$$q(j \mid i) = \beta P_{ij} \tag{6}$$

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 state of the world 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) 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 (5) are

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

for all s_t, s_{t+1}

or, under our assumption (6) about the values taken by Arrow security prices,

$$c_{t+1} = c_t \tag{7}$$

Thus, our consumer sets $c_t = \bar{c}$ for all $t \geq 0$ for some value $\bar{c}$ that it is our job now to determine along with values for $b_{t+1}(j|s_t = i)$ for $i = 1, 2$ and $j = 1, 2$

We'll use a *guess and verify* method to determine these objects

Guess: We'll make the plausible guess that

$$b_{t+1}(s_{t+1} = j \,|\, s_t = i) = b(j), \quad i = 1, 2; \;\; j = 1, 2 \tag{8}$$

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 (8) about the consumer's debt level choices.

For $t \geq 1$, these imply

$$\begin{aligned} \bar{c} + b(1) &= y(1) + q(1 \,|\, 1)b(1) + q(2 \,|\, 1)b(2) \\ \bar{c} + b(2) &= y(2) + q(1 \,|\, 2)b(1) + q(2 \,|\, 2)b(2) \end{aligned} \tag{9}$$

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

$$\bar{c} + b_0 = y(1) + q(1 \,|\, 1)b(1) + q(2 \,|\, 1)b(2) \tag{10}$$

where b_0 is the (exogenous) debt the consumer is assumed to bring into period 0

If we substitute (10) into the first equation of (9) and rearrange, we discover that

$$b(1) = b_0 \tag{11}$$

We can then use the second equation of (9) to deduce the restriction

$$y(1) - y(2) + [q(1\,|\,1) - q(1\,|\,2) - 1]b_0 + [q(2\,|\,1) + 1 - q(2\,|\,2)]b(2) = 0, \tag{12}$$

an equation that we can solve for the unknown $b(2)$.

Knowing $b(1)$ and $b(2)$, we can solve equation (10) for the constant level of consumption $\bar{c}$.

56.5.1 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 (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.

56.5.2 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

```
In [4]: 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

    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)
    db = ((1 - β) * v - y) / β

    for i, s in enumerate(s_path):
        c_path[i] = (1 - β) * (v - b_path[i] * np.ones((n, 1)))[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

```
In [5]: cp = ConsumptionProblem()
        c_bar, b = consumption_complete(cp)
        np.isclose(c_bar + b[1] - cp.y[1] - (cp.β * cp.P)[1, :] @ b, 0)
```

```
Out[5]: 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 [67]

56.6 Model 2 (One-Period Risk-Free Debt Only)

This is a version of the original models of Hall (1978) in which the consumer's ability to substitute intertemporally is constrained by his ability to buy or sell only one security, a risk-free one-period bond bearing a constant gross interest rate that equals β^{-1}.

Given an initial debt b_0 at time 0, the consumer faces a sequence of budget constraints

$$c_t + b_t = y_t + \beta b_{t+1}, \quad t \geq 0$$

where β is the price at time t of a risk-free claim on one unit of time consumption at time $t+1$.

First-order conditions for the consumer's problem are

$$\sum_j u'(c_{t+1,j}) P_{ij} = u'(c_{t,i})$$

For our assumed quadratic utility function this implies

$$\sum_j c_{t+1,j} P_{ij} = c_{t,i} \tag{13}$$

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

$$b_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - (1-\beta)^{-1} c_t \tag{14}$$

and

$$c_t = (1-\beta) \left[\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - b_t \right] \tag{15}$$

Equation (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 (15) into the one-period budget constraint and rearranging leads to

$$b_{t+1} - b_t = \beta^{-1} \left[(1-\beta) \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - y_t \right] \tag{16}$$

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

$$v_t := \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$$

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 (15), consumption at date t when debt is b_t and the Markov state today is $s_t = i$ is evidently

$$c(b_t, i) = (1 - \beta) \left([(I - \beta P)^{-1} \vec{y}]_i - b_t \right) \tag{17}$$

and the increment to debt is

$$b_{t+1} - b_t = \beta^{-1} [(1 - \beta) v(i) - y(i)] \tag{18}$$

56.6.1 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.

56.6.2 The Incomplete Markets Model

The code above also contains a function called consumption_incomplete() that uses (17) and (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

In [6]:
```
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=(15, 5))

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), c_bar * np.ones(N_simul),
                    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.

56.6.3 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 [14] as well as in the complete markets tradition of Lucas and Stokey [111].

Chapter 57

Tax Smoothing with Complete and Incomplete Markets

57.1 Contents

- Overview 57.2
- Tax Smoothing with Complete Markets 57.3
- Returns on State-Contingent Debt 57.4
- More Finite Markov Chain Tax-Smoothing Examples 57.5

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

```
In [1]: !pip install --upgrade quantecon
```

57.2 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 [111].
- the other is in the **incomplete markets** tradition of Barro [14].

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 [14] 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 [67] had assumed an exogenous stochastic process of nonfinancial income and an exogenous gross interest rate on one period risk-free debt that equals β^{-1}, where $\beta \in (0, 1)$ is also a consumer's intertemporal discount factor.

Barro [14] 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 [111].

57.2.1 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) [61] seems to have prescribed policies that come from Barro's model [14]

Let's start with some standard imports:

```
In [2]: 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

57.2.2 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

```
In [3]: 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

    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)
    db = ((1 - β) * v - y) / β

    for i, s in enumerate(s_path):
        c_path[i] = (1 - β) * (v - b_path[i] * np.ones((n, 1)))[s, 0]
```

```
        b_path[i + 1] = b_path[i] + db[s, 0]

    return c_path, b_path[:-1], y[s_path]
```

57.2.3 Revisiting the consumption-smoothing model

The code above also contains a function called consumption_incomplete() that uses (17) and (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

```
In [4]: 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=(15, 5))

        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), c_bar * np.ones(N_simul), 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.

```
In [5]: fig, ax = plt.subplots(1, 2, figsize=(15, 5))

        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), c_bar * np.ones(N_simul), 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()
```

57.3 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 [111], 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.

57.4 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.

In [6]:
```
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

    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
```

57.4.1 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.

In [7]:
```
# 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)
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)

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
```

57.4.2 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 $b_2 = -10$.

57.5 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.

```
In [8]: 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

                # 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), self.T_bar * np.ones(N), 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

            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])
```

57.5.1 Parameters

```
In [9]: γ = .1
        λ = .1
        ϕ = .1
        θ = .1
        ψ = .1
        g_L = .5
        g_M = .8
        g_H = 1.2
        β = .96
```

57.5.2 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.)

```
In [10]: 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']
```

```
In [11]: ts_ex1 = TaxSmoothingExample(g_ex1, P_ex1, b0_ex1, states_ex1,
 ↪random_state=1)
         ts_ex1.display()
```

```
    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

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
```

```
In [12]: # 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()
```

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
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Periods

Government debt paths
1
0
−1
−2
−3
−4
incomplete market
complete market
govt expenditures
value of debts today
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
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

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
```

57.5.3 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.

```
In [13]: g_ex2 = [g_L, g_L, g_H]
         P_ex2 = np.array([[1,    0,     0],
                           [0, 1-γ,     γ],
                           [ϕ,    0, 1-ϕ]])
         b0_ex2 = 1
         states_ex2 = ['peace', 'temporary peace', 'war']
```

```
In [14]: ts_ex2 = TaxSmoothingExample(g_ex2, P_ex2, b0_ex2, states_ex2, init=1,
 ↪random_state=1)
         ts_ex2.display()
```

Government debt paths
2
1
0
−1
−2
−3
−4
0
10
20
30
40
50
60
70
80
Periods
incomplete market
complete market
govt expenditures
value of debts today

Cumulative return path (complete markets)
Cumulative return
0
−2
−4
−6
−8
Government 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

```
P
[[1.  0.  0. ]
```

```
 [0.  0.9 0.1]
 [0.1 0.  0.9]]
Q
 [[0.96  0.    0.   ]
 [0.    0.864 0.096]
 [0.096 0.    0.864]]
Govt expenditures in peace, temporary peace, war = [0.5 0.5 1.2]
Constant tax collections = 0.6053287197231834
Govt debt in 3 states = [ 2.63321799 -1.         -2.51384083]

Government tax collections minus debt levels in peace, temporary peace, war
  T+b in peace = -2.027889273356399
  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
-9.3689917325942
```

57.5.4 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 $s_0 = 1$.

```
In [15]: g_ex3 = [g_L, g_L, g_H, g_H]
```

```
P_ex3 = np.array([[1-λ,  λ,    0,     0],
                  [0,  1-ϕ,    ϕ,     0],
                  [0,    0,  1-ψ,     ψ],
                  [θ,    0,    0,   1-θ ]])
b0_ex3 = 1
states_ex3 = ['peace1', 'peace2', 'war1', 'war2']
```

```
In [16]: 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]]
```

```
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
```

57.5.5 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 $s_0 = 1$.

```
In [17]: 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']
```

```
In [18]: 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]
```

```
 [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
```

57.5.6 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 $s_0 = 1$.

```
In [19]: 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']
```

```
In [20]: ts_ex5 = TaxSmoothingExample(g_ex5, P_ex5, b0_ex5, states_ex5, N_simul=7,
         random_state=1)
         ts_ex5.display()
```

```
   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

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
```

57.5.7 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 [14].

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 58

Robustness

58.1 Contents

- Overview 58.2
- The Model 58.3
- Constructing More Robust Policies 58.4
- Robustness as Outcome of a Two-Person Zero-Sum Game 58.5
- The Stochastic Case 58.6
- Implementation 58.7
- Application 58.8
- Appendix 58.9

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

```
In [1]: !pip install --upgrade quantecon
```

58.2 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:

```
In [2]: import pandas as pd
        import numpy as np
        from scipy.linalg import eig
        import matplotlib.pyplot as plt
        %matplotlib inline
        import quantecon as qe
```

58.2.1 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.

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).

58.2.2 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.

58.2.3 Other References

Our discussion in this lecture is based on

- [75]
- [71]

58.3 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

$$\sum_{t=0}^{\infty} \beta^t \{x_t' R x_t + u_t' Q u_t\} \tag{1}$$

subject to the linear law of motion

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

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 $\sum_{t=0}^{\infty} \beta^{t+1} w_{t+1}' w_{t+1}$.

58.4 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

$$J(x) = \min_u \max_w \{x'Rx + u'Qu + \beta\,[J(Ax + Bu + Cw) - \theta w'w]\} \tag{3}$$

Notice that we've added the penalty term $-\theta w'w$.

Since $w'w = \|w\|^2$, this term becomes influential when w moves away from the origin.

The penalty parameter θ controls how much we penalize the maximizing agent for "harming" the minimizing agent.

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

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

58.4.1 Analyzing the Bellman Equation

So what does J in (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

$$\begin{aligned} \max_w \{(Ax + Bu + Cw)'P(Ax + Bu + Cw) - \theta w'w\} \\ = (Ax + Bu)'\mathcal{D}(P)(Ax + Bu) \end{aligned} \tag{4}$$

where

$$\mathcal{D}(P) := P + PC(\theta I - C'PC)^{-1}C'P \tag{5}$$

and I is a $j \times j$ identity matrix. Substituting this expression for the maximum into (3) yields

$$x'Px = \min_u \{x'Rx + u'Qu + \beta\,(Ax + Bu)'\mathcal{D}(P)(Ax + Bu)\} \tag{6}$$

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 (6) and working through the algebra gives $x' P x = x' \mathcal{B}(\mathcal{D}(P)) x$ for all x, or, equivalently,

$$P = \mathcal{B}(\mathcal{D}(P))$$

where $\mathcal{D}$ is the operator defined in (5) and

$$\mathcal{B}(P) := R - \beta^2 A' P B (Q + \beta B' P B)^{-1} B' P A + \beta A' P A$$

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 [71]), the operator $\mathcal{B} \circ \mathcal{D}$ has a unique positive definite fixed point, which we denote below by $\hat{P}$.

A robust policy, indexed by θ, is $u = -\hat{F} x$ where

$$\hat{F} := (Q + \beta B' \mathcal{D}(\hat{P}) B)^{-1} \beta B' \mathcal{D}(\hat{P}) A \tag{7}$$

We also define

$$\hat{K} := (\theta I - C' \hat{P} C)^{-1} C' \hat{P} (A - B \hat{F}) \tag{8}$$

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 (4) evaluated at the fixed rule $u = -\hat{F} x$.

Note that $\hat{P}, \hat{F}, \hat{K}$ are all determined by the primitives and θ.

Note also that if θ is very large, then $\mathcal{D}$ is approximately equal to the identity mapping.

Hence, when θ is large, $\hat{P}$ and $\hat{F}$ are approximately equal to their standard LQ values.

Furthermore, when θ is large, $\hat{K}$ is approximately equal to zero.

Conversely, smaller θ is associated with greater fear of model misspecification and greater concern for robustness.

58.5 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.

58.5.1 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

$$\sum_{t=1}^{\infty} \beta^t w_t' w_t \leq \eta \tag{9}$$

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 (1), this value can be written as

$$J_F(x_0, \mathbf{w}) := \sum_{t=0}^{\infty} \beta^t x_t' (R + F'QF) x_t \tag{10}$$

where

$$x_{t+1} = (A - BF)x_t + Cw_{t+1} \tag{11}$$

and the initial condition x_0 is as specified in the left side of (10).

Agent 2 chooses $\mathbf{w}$ to maximize agent 1's loss $J_F(x_0, \mathbf{w})$ subject to (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 (11) and θ is a Lagrange multiplier on constraint (9).

For the moment, let's take θ as fixed, allowing us to drop the constant $\beta\theta\eta$ term in the objective function, and hence write the problem as

$$\max_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \left\{ x_t' (R + F'QF) x_t - \beta\theta w_{t+1}' w_{t+1} \right\}$$

or, equivalently,

$$\min_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \left\{ -x_t' (R + F'QF) x_t + \beta\theta w_{t+1}' w_{t+1} \right\} \tag{12}$$

subject to (11).

What's striking about this optimization problem is that it is once again an LQ discounted dynamic programming problem, with $\mathbf{w} = \{w_t\}$ as the sequence of controls.

The expression for the optimal policy can be found by applying the usual LQ formula (see here).

We denote it by $K(F, \theta)$, with the interpretation $w_{t+1} = K(F, \theta)x_t$.

The remaining step for agent 2's problem is to set θ to enforce the constraint (9), which can be done by choosing $\theta = \theta_\eta$ such that

$$\beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \theta_\eta)' K(F, \theta_\eta) x_t = \eta \tag{13}$$

Here x_t is given by (11) — which in this case becomes $x_{t+1} = (A - BF + CK(F, \theta))x_t$.

58.5.2 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 (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

$$R_\theta(x_0, F) - \theta \text{ ent} \leq \sum_{t=0}^{\infty} \beta^t \left\{-x_t'(R + F'QF)x_t\right\} \tag{14}$$

where

$$\text{ent} := \beta \sum_{t=0}^{\infty} \beta^t w_{t+1}' w_{t+1}$$

The left side of inequality (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 (14) is attained when

$$\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \theta)' K(F, \theta) x_t \tag{15}$$

To construct the *lower bound* on the set of values associated with all perturbations $\mathbf{w}$ satisfying the entropy constraint (9) at a given entropy level, we proceed as follows:

- For a given θ, solve the minimization problem (12).
- Compute the minimizer $R_\theta(x_0, F)$ and the associated entropy using (15).
- Compute the lower bound on the value function $R_\theta(x_0, F) - \theta$ ent and plot it against ent.
- Repeat the preceding three steps for a range of values of θ to trace out the lower bound.

Note

This procedure sweeps out a set of separating hyperplanes indexed by different values for the Lagrange multiplier θ.

The Upper Bound

To construct an *upper bound* we use a very similar procedure.

We simply replace the *minimization* problem (12) with the *maximization* problem

$$V_{\tilde\theta}(x_0, F) = \max_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t - \beta\tilde\theta w_{t+1}' w_{t+1} \right\} \tag{16}$$

where now $\tilde\theta > 0$ penalizes the choice of $\mathbf{w}$ with larger entropy.

(Notice that $\tilde\theta = -\theta$ in problem (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

$$V_{\tilde\theta}(x_0, F) + \tilde\theta \text{ ent} \geq \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t \right\} \tag{17}$$

where

$$\text{ent} \equiv \beta \sum_{t=0}^{\infty} \beta^t w_{t+1}' w_{t+1}$$

The left side of inequality (17) is a straight line with slope $\tilde\theta$.

The upper bound on the left side of (17) is attained when

$$\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \tilde\theta)' K(F, \tilde\theta) x_t \tag{18}$$

To construct the *upper bound* on the set of values associated all perturbations $\mathbf{w}$ with a given entropy we proceed much as we did for the lower bound

  - For a given $\tilde{\theta}$, solve the maximization problem (16).

- Compute the maximizer $V_{\tilde{\theta}}(x_0, F)$ and the associated entropy using (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.

58.5.3 Agent 1's Problem

Now we turn to agent 1, who solves

$$\min_{\{u_t\}} \sum_{t=0}^{\infty} \beta^t \left\{ x_t' R x_t + u_t' Q u_t - \beta \theta w_{t+1}' w_{t+1} \right\} \tag{19}$$

where $\{w_{t+1}\}$ satisfies $w_{t+1} = K x_t$.

In other words, agent 1 minimizes

$$\sum_{t=0}^{\infty} \beta^t \left\{ x_t' (R - \beta \theta K' K) x_t + u_t' Q u_t \right\} \tag{20}$$

subject to

$$x_{t+1} = (A + CK) x_t + B u_t \tag{21}$$

Once again, the expression for the optimal policy can be found here — we denote it by $\tilde{F}$.

58.5.4 Nash Equilibrium

Clearly, the $\tilde{F}$ we have obtained depends on K, which, in agent 2's problem, depended on an initial policy F.

Holding all other parameters fixed, we can represent this relationship as a mapping Φ, where

$$\tilde{F} = \Phi(K(F, \theta))$$

The map $F \mapsto \Phi(K(F, \theta))$ corresponds to a situation in which

1. agent 1 uses an arbitrary initial policy F
2. agent 2 best responds to agent 1 by choosing $K(F, \theta)$
3. agent 1 best responds to agent 2 by choosing $\tilde{F} = \Phi(K(F, \theta))$

As you may have already guessed, the robust policy $\hat{F}$ defined in (7) is a fixed point of the mapping Φ.

In particular, for any given θ,

1. $K(\hat{F}, \theta) = \hat{K}$, where $\hat{K}$ is as given in (8)
2. $\Phi(\hat{K}) = \hat{F}$

A sketch of the proof is given in the appendix.

58.6 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 (3) with the stochastic analog

$$J(x) = \min_u \max_{\psi \in \mathcal{P}} \left\{ x' R x + u' Q u + \beta \left[\int J(Ax + Bu + Cw) \, \psi(dw) - \theta D_{KL}(\psi, \phi) \right] \right\} \tag{22}$$

Here $\mathcal{P}$ represents the set of all densities on $\mathbb{R}^n$ and ϕ is the benchmark distribution $N(0, I)$.

The distribution ϕ is chosen as the least desirable conditional distribution in terms of next period outcomes, while taking into account the penalty term $\theta D_{KL}(\psi, \phi)$.

This penalty term plays a role analogous to the one played by the deterministic penalty $\theta w' w$ in (3), since it discourages large deviations from the benchmark.

58.6.1 Solving the Model

The maximization problem in (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

$$\begin{aligned} \max_{\psi \in \mathcal{P}} \left\{ \int (Ax + Bu + Cw)'P(Ax + Bu + Cw)\, \psi(dw) - \theta D_{KL}(\psi, \phi) \right\} \\ = (Ax + Bu)'\mathcal{D}(P)(Ax + Bu) + \kappa(\theta, P) \end{aligned} \tag{23}$$

where

$$\kappa(\theta, P) := \theta \ln[\det(I - \theta^{-1}C'PC)^{-1}]$$

and the maximizer is the Gaussian distribution

$$\psi = N\left((\theta I - C'PC)^{-1}C'P(Ax + Bu), (I - \theta^{-1}C'PC)^{-1}\right) \tag{24}$$

Substituting the expression for the maximum into Bellman equation (22) and using $J(x) = x'Px + d$ gives

$$x'Px + d = \min_u \{x'Rx + u'Qu + \beta\,(Ax + Bu)'\mathcal{D}(P)(Ax + Bu) + \beta\,[d + \kappa(\theta, P)]\} \tag{25}$$

Since constant terms do not affect minimizers, the solution is the same as (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

$$\hat{d} := \frac{\beta}{1 - \beta}\kappa(\theta, P) \tag{26}$$

The robust policy in this stochastic case is the minimizer in (25), which is once again $u = -\hat{F}x$ for $\hat{F}$ given by (7).

Substituting the robust policy into (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 (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.

58.6.2 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 (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 (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

$$d_F := \frac{\beta}{1 - \beta} \kappa(\theta, P_F) = \frac{\beta}{1 - \beta} \theta \ln[\det(I - \theta^{-1} C' P_F C)^{-1}] \tag{27}$$

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.

58.7 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 (7) – (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 (13)
- `evaluate_F()` computes the loss and entropy associated with a given policy — see this discussion

58.8 Application

Let us consider a monopolist similar to this one, but now facing model uncertainty.

The inverse demand function is $p_t = a_0 - a_1 y_t + d_t$.

where

$$d_{t+1} = \rho d_t + \sigma_d w_{t+1}, \quad \{w_t\} \overset{\text{IID}}{\sim} N(0,1)$$

and all parameters are strictly positive.

The period return function for the monopolist is

$$r_t = p_t y_t - \gamma \frac{(y_{t+1} - y_t)^2}{2} - c y_t$$

Its objective is to maximize expected discounted profits, or, equivalently, to minimize $\mathbb{E} \sum_{t=0}^{\infty} \beta^t (-r_t)$.

To form a linear regulator problem, we take the state and control to be

$$x_t = \begin{bmatrix} 1 \\ y_t \\ d_t \end{bmatrix} \quad \text{and} \quad u_t = y_{t+1} - y_t$$

Setting $b := (a_0 - c)/2$ we define

$$R = -\begin{bmatrix} 0 & b & 0 \\ b & -a_1 & 1/2 \\ 0 & 1/2 & 0 \end{bmatrix} \quad \text{and} \quad Q = \gamma/2$$

For the transition matrices, we set

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \rho \end{bmatrix}, \qquad B = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \qquad C = \begin{bmatrix} 0 \\ 0 \\ \sigma_d \end{bmatrix}$$

Our aim is to compute the value-entropy correspondences shown above.

The parameters are

$$a_0 = 100, a_1 = 0.5, \rho = 0.9, \sigma_d = 0.05, \beta = 0.95, c = 2, \gamma = 50.0$$

The standard normal distribution for w_t is understood as the agent's baseline, with uncertainty parameterized by θ.

We compute value-entropy correspondences for two policies

1. The no concern for robustness policy F_0, which is the ordinary LQ loss minimizer.
2. A "moderate" concern for robustness policy F_b, with $\theta = 0.02$.

The code for producing the graph shown above, with blue being for the robust policy, is as follows

In [3]:
```
"""
Authors: Chase Coleman, Spencer Lyon, Thomas Sargent, John Stachurski
"""

# 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],
              [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
    =======
    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 ϑ
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 ϑ exceeds the breakdown point
test_matrix = np.identity(Pb.shape[0]) - (C.T @ Pb @ C) / θ
eigenvals, eigenvecs = eig(test_matrix)
assert (eigenvals >= 0).all(), 'θ below breakdown point.'


emax = 1.6e6

optimal_best_case = value_and_entropy(emax, Fo, 'best')
robust_best_case = value_and_entropy(emax, Fb, 'best')
optimal_worst_case = value_and_entropy(emax, Fo, 'worst')
robust_worst_case = value_and_entropy(emax, Fb, 'worst')

fig, ax = plt.subplots()

ax.set_xlim(0, emax)
ax.set_ylabel("Value")
ax.set_xlabel("Entropy")
ax.grid()

for axis in 'x', 'y':
    plt.ticklabel_format(style='sci', axis=axis, scilimits=(0, 0))

plot_args = {'lw': 2, 'alpha': 0.7}

colors = 'r', 'b'

df_pairs = ((optimal_best_case, optimal_worst_case),
            (robust_best_case, robust_worst_case))


class Curve:

    def __init__(self, x, y):
        self.x, self.y = x, y

    def __call__(self, z):
        return np.interp(z, self.x, self.y)


for c, df_pair in zip(colors, df_pairs):
    curves = []
    for df in df_pair:
        # Plot curves
        x, y = df['entropy'], df['value']
        x, y = (np.asarray(a, dtype='float') for a in (x, y))
        egrid = np.linspace(0, emax, 100)
        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 0x7fd12b45ac50>]
[<matplotlib.lines.Line2D object at 0x7fd12b3f5128>]
[<matplotlib.lines.Line2D object at 0x7fd12b45ab70>]
[<matplotlib.lines.Line2D object at 0x7fd12b3f5b38>]
```

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?

58.9 Appendix

We sketch the proof only of the first claim in this section, which is that, for any given θ, $K(\hat{F}, \theta) = \hat{K}$, where $\hat{K}$ is as given in (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 (7), then

$$K(\hat{F}, \theta) = (\theta I - C'\hat{P}C)^{-1} C'\hat{P}(A - B\hat{F}) \tag{28}$$

Proof: As a first step, observe that when $F = \hat{F}$, the Bellman equation associated with the LQ problem (11) – (12) is

$$\tilde{P} = -R - \hat{F}'Q\hat{F} - \beta^2(A - B\hat{F})'\tilde{P}C(\beta\theta I + \beta C'\tilde{P}C)^{-1}C'\tilde{P}(A - B\hat{F}) + \beta(A - B\hat{F})'\tilde{P}(A - B\hat{F}) \tag{29}$$

(revisit this discussion if you don't know where (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 (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 (28).

Hence it remains only to show that $-\hat{P}$ solves (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 59

Markov Jump Linear Quadratic Dynamic Programming

59.1 Contents

- Overview 59.2
- Review of useful LQ dynamic programming formulas 59.3
- Linked Ricatti equations for Markov LQ dynamic programming 59.4
- Applications 59.5
- Example 1 59.6
- Example 2 59.7
- More examples 59.8

Co-authors: Sebastian Graves and Zejin Shi

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

```
In [1]: !pip install --upgrade quantecon
```

59.2 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 [46] and the references cited there.

The method has been applied to problems in macroeconomics and monetary economics by [155] and [154].

The periodic models of seasonality described in chapter 14 of [78] 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 pro-**

gramming 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.

59.3 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 ρ satisfies

$$\rho = \beta\left(\rho + \text{trace}(PCC')\right)$$

and the matrix F in the decision rule for u_t satisfies

$$F = (Q + \beta B'PB)^{-1}(\beta(B'PA) + W)$$

With the preceding formulas in mind, we are ready to approach Markov Jump linear quadratic dynamic programming.

59.4 Linked Ricatti 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 Π having typical element Π_{ij}.

Here i denotes today and j denotes tomorrow and

$$\Pi_{ij} = \text{Prob}(s_{t+1} = j | s_t = i)$$

We'll switch between labeling today's state as s_t and i and between labeling tomorrow's state as s_{t+1} or j.

The decision-maker solves the minimization problem:

$$\min_{\{u_t\}_{t=0}^{\infty}} E \sum_{t=0}^{\infty} \beta^t r(x_t, s_t, u_t)$$

with

$$r(x_t, s_t, u_t) = -(x_t' R_{s_t} x_t + u_t' Q_{s_t} u_t + 2u_t' W_{s_t} x_t)$$

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

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

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

$$F_i = (Q_i + \beta \sum_j \Pi_{ij} B_i' P_j B_i)^{-1} (\beta \sum_j \Pi_{ij} (B_i' P_j A_i) + W_i)$$

59.5 Applications

We now describe some Python code and a few examples that put the code to work.

To begin, we import these Python modules

In [2]:
```
import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

%matplotlib inline
```

In [3]:
```
# Set discount factor
β = 0.95
```

59.6 Example 1

This example is a version of a classic problem of optimally adjusting a variable k_t to a target level in the face of costly adjustment.

This provides a model of gradual adjustment.

Given k_0, the objective function is

$$\max_{\{k_t\}_{t=1}^{\infty}} E_0 \sum_{t=0}^{\infty} \beta^t r\left(s_t, k_t\right)$$

where the one-period payoff function is

$$r(s_t, k_t) = f_{1,s_t} k_t - f_{2,s_t} k_t^2 - d_{s_t}(k_{t+1} - k_t)^2,$$

E_0 is a mathematical expectation conditioned on time 0 information x_0, s_0 and the transition law for continuous state variable k_t is

$$k_{t+1} - k_t = u_t$$

We can think of k_t as the decision-maker's capital and u_t as costs of adjusting the level of capital.

We assume that $f_1\left(s_t\right) > 0$, $f_2\left(s_t\right) > 0$, and $d\left(s_t\right) > 0$.

Denote the state transition matrix for Markov state $s_t \in \{1, 2\}$ as Π:

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

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

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

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

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

```
In [4]: 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.

```
In [5]: 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

In [6]:
```
# Construct Markov transition matrix
Π1 = np.array([[0., 1.],
               [1., 0.]])
```

In [7]:
```
# Construct matrices
Qs, Rs, Ns, As, Bs, Cs, k_star = construct_arrays1(d_vals=[1., 0.5])
```

In [8]:
```
# 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

In [9]:
```
# P(s)
ex1_a.Ps
```

Out[9]:
```
array([[[ 1.56626026, -0.78313013],
        [-0.78313013, -4.60843493]],

       [[ 1.37424214, -0.68712107],
        [-0.68712107, -4.65643947]]])
```

In [10]:
```
# d(s) = 0, since there is no randomness
ex1_a.ds
```

Out[10]:
```
array([0., 0.])
```

In [11]:
```
# F(s)
ex1_a.Fs
```

Out[11]:
```
array([[[ 0.56626026, -0.28313013]],

        [[ 0.74848427, -0.37424214]]])
```

Now we'll plot the decision rules and see if they make sense

In [12]:
```
# 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)")

# 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.

In [13]:
```
# 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()
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

$$\Pi_2 = \begin{bmatrix} 1-\lambda & \lambda \\ \lambda & 1-\lambda \end{bmatrix}.$$

In [14]:
```
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
```

Out[14]:
```
array([[[ 0.57291724, -0.28645862]],

       [[ 0.74434525, -0.37217263]]])
```

In [15]:
```
λ = 0.2 # low λ
Π2 = np.array([[1-λ, λ],
               [λ, 1-λ]])

ex1_b = qe.LQMarkov(Π2, Qs, Rs, As, Bs, Cs=Cs, Ns=Ns, beta=β)
ex1_b.stationary_values();
ex1_b.Fs
```

Out[15]:
```
array([[[ 0.59533259, -0.2976663 ]],

       [[ 0.72818728, -0.36409364]]])
```

We can plot optimal decision rules associated with different λ values.

```
In [16]: 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, :]
```

```
In [17]: 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 λ.

Evidently, as the Markov chain becomes *more nearly periodic* (i.e., as $\lambda \to 1$), the dynamic program adjusts capital faster in the low adjustment cost Markov state to take advantage of what is only temporarily a more favorable time to invest.

Now let's study situations in which the Markov transition matrix Π is asymmetric

$$\Pi_3 = \begin{bmatrix} 1-\lambda & \lambda \\ \delta & 1-\delta \end{bmatrix}.$$

In [18]:
```
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
```

Out[18]:
```
array([[[ 0.57169781, -0.2858489 ]],

       [[ 0.72749075, -0.36374537]]])
```

We can plot optimal decision rules for different λ and δ values.

In [19]:
```
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, :]
```

```
In [20]: 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

```
In [21]: def run(construct_func, vals_dict, state_vec):
             """
             A Wrapper function that repeats the computation above
             for different cases
             """

             Qs, Rs, Ns, As, Bs, Cs, k_star = construct_func(**vals_dict)

             # Symmetric Π
             # Notice that pure periodic transition is a special case
             # when λ=1
             print("symmetric Π case:\n")
             λ_vals = np.linspace(0., 1., 10)
             F1 = np.empty((λ_vals.size, len(state_vec)))
             F2 = np.empty((λ_vals.size, len(state_vec)))

             for i, λ in enumerate(λ_vals):
                 Π2 = np.array([[1-λ, λ],
                                [λ, 1-λ]])

                 mplq = qe.LQMarkov(Π2, Qs, Rs, As, Bs, Cs=Cs, Ns=Ns, beta=β)
                 mplq.stationary_values();
                 F1[i, :] = mplq.Fs[0, 0, :]
                 F2[i, :] = mplq.Fs[1, 0, :]

             for i, state_var in enumerate(state_vec):
                 fig = plt.figure()
                 ax = fig.add_subplot(111)
                 ax.plot(λ_vals, F1[:, i], label="$\overline{s}_1$", 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")
    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 λ.

We call $k^{target}_{s_t}$ a "target" because in each Markov state s_t, optimal policies are contraction mappings and will push k_t towards a fixed point $k^{target}_{s_t}$.

When $\lambda \to 0$, each Markov state becomes close to absorbing state and consequently $k^{target}_{s_t} \to k^*_{s_t}$.

But when $\lambda \to 1$, the Markov transition matrix becomes more nearly periodic, so the optimum decision rules target more at the optimal k level in the other state in order to enjoy higher expected payoff in the next period.

The switch happens at $\lambda = 0.5$ when both states are equally likely to be reached.

Below we plot an additional figure that shows optimal k levels in the two states Markov jump state and also how the targeted k levels change as λ changes.

```
In [22]: run(construct_arrays1, {"f1_vals":[0.5, 1.]}, state_vec1)
```

```
symmetric 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
λ
$\bar{s}_1$
$\bar{s}_2$

coefficient on constant term
−0.16
−0.18
−0.20
−0.22
−0.24
−0.26
−0.28
−0.30
$F(\bar{s}_t)$
0.0
0.2
0.4
0.6
0.8
1.0
λ
$\bar{s}_1$
$\bar{s}_2$

asymmetric Π case:

Set f_{1,s_t} and d_{s_t} as constant functions and

$$f_{2,1} = 0.5, f_{2,2} = 1$$

```
In [23]: run(construct_arrays1, {"f2_vals":[0.5, 1.]}, state_vec1)
```

```
symmetric Π case:
```

asymmetric Π case:

59.7 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

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

```
In [24]: 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
                 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
```

```
In [25]: state_vec2 = ["k", "constant term", "w"]
```

Only d_{s_t} depends on s_t.

```
In [26]: run(construct_arrays2, {"d_vals":[1., 0.5]}, state_vec2)
```

```
symmetric Π case:
```

asymmetric Π case:

coefficient on k
0.750
0.725
0.700
0.675
0.650
0.625
0.600
0.575
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
θ

coefficient on constant term
0.18
0.16
0.14
0.12
0.10
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
θ

Only f_{1,s_t} depends on s_t.

```
In [27]: 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
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
δ

coefficient on constant term
0.30
0.28
0.26
0.24
0.22
0.20
0.18
0.16
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
δ

Only f_{2,s_t} depends on s_t.

```
In [28]: 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
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
θ

coefficient on constant term
0.35
0.30
0.25
0.20
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
θ

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

```
In [29]: 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
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
δ

coefficient on constant term
0.15
0.10
0.05
0.00
−0.05
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
δ

Only ρ_{s_t} depends on s_t.

```
In [30]: 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
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
σ

coefficient on constant term
0.15
0.14
0.13
0.12
0.11
0.10
0.09
0.08
0.07
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
σ

Only σ_{s_t} depends on s_t.

```
In [31]: 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
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
θ

coefficient on constant term
5.5
5.0
4.5
4.0
3.5
3.0
0.0
0.2
0.4
0.6
0.8
1.0
λ
0.0
0.2
0.4
0.6
0.8
1.0
θ

59.8 More examples

The following lectures describe how Markov jump linear quadratic dynamic programming can be used to extend the [14] model of optimal tax-smoothing and government debt in several interesting directions

1. How to Pay for a War: Part 1

1. How to Pay for a War: Part 2
2. How to Pay for a War: Part 3

Chapter 60

How to Pay for a War: Part 1

60.1 Contents

- Reader's Guide 60.2
- Public Finance Questions 60.3
- Barro (1979) Model 60.4
- Python Class to Solve Markov Jump Linear Quadratic Control Problems 60.5
- Barro Model with a Time-varying Interest Rate 60.6

Co-author: Sebastian Graves

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

```
In [1]: !pip install --upgrade quantecon
```

60.2 Reader's Guide

Let's start with some standard imports:

```
In [2]: 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 [14] 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

1. How to Pay for a War: Part 3

The extensions are modified versions of his 1979 model later suggested by Barro (1999 [15], 2003 [16]).

Barro's original 1979 [14] 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 [14] 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 [14] model along lines he suggested in Barro (1999 [15], 2003 [16]).

Barro (1979) [14] 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 β^{-1}, the reciprocal of the factor β at which the government discounts future tax distortions.

Barro's model can be mapped into a discounted linear quadratic dynamic programming problem.

Partly inspired by Barro (1999) [15] and Barro (2003) [16], our generalizations of Barro's (1979) [14] 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.

60.3 Public Finance Questions

Barro's 1979 [14] 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 [15] and 2003 [16] 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.

60.4 Barro (1979) Model

We begin by solving a version of the Barro (1979) [14] 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, β a discount factor, $b_{t,t+1}$ time $t+1$ goods that the government promises to pay at t, G_t government purchases, $p_{t,t+1}$ the number of time t goods received per time $t+1$ goods promised.

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

In the spirit of Barro (1979) [14], the stochastic process of government expenditures is exogenous.

The government's problem is to choose a plan for taxation and borrowing $\{b_{t+1}, T_t\}_{t=0}^{\infty}$ to minimize

$$E_0 \sum_{t=0}^{\infty} \beta^t T_t^2$$

subject to the constraints

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

$$G_t = U_g z_t$$

$$z_{t+1} = A_{22} z_t + C_2 w_{t+1}$$

where $w_{t+1} \sim N(0, I)$

The variables $T_t, b_{t,t+1}$ are *control* variables chosen at t, while $b_{t-1,t}$ is an endogenous state variable inherited from the past at time t and $p_{t,t+1}$ is an exogenous state variable at time t.

To begin, we assume that $p_{t,t+1}$ is constant (and equal to β)

- later we will extend the model to allow $p_{t,t+1}$ to vary over time

To map into the LQ framework, we use $x_t = \begin{bmatrix} b_{t-1,t} \\ z_t \end{bmatrix}$ as the state vector, and $u_t = b_{t,t+1}$ as the control variable.

Therefore, the (A, B, C) matrices are defined by the state-transition law:

$$x_{t+1} = \begin{bmatrix} 0 & 0 \\ 0 & A_{22} \end{bmatrix} x_t + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_t + \begin{bmatrix} 0 \\ C_2 \end{bmatrix} w_{t+1}$$

To find the appropriate (R, Q, W) matrices, we note that G_t and $b_{t-1,t}$ can be written as appropriately defined functions of the current state:

$$G_t = S_G x_t \ , \ b_{t-1,t} = S_1 x_t$$

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:

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

In [3]:
```
# Model parameters
β, Gbar, ρ, σ = 0.95, 5, 0.8, 1

# Basic model matrices
A22 = np.array([[1,    0],
                [Gbar, ρ],])

C2 = np.array([[0],
               [σ]])

Ug = np.array([[0, 1]])

# LQ framework matrices
A_t = np.zeros((1, 3))
A_b = np.hstack((np.zeros((2, 1)), A22))
A = np.vstack((A_t, A_b))

B = np.zeros((3, 1))
B[0, 0] = 1

C = np.vstack((np.zeros((1, 1)), C2))

Sg = np.hstack((np.zeros((1, 1)), Ug))
S1 = np.zeros((1, 3))
S1[0, 0] = 1
S = S1 + Sg

M = np.array([[-β]])

R = S.T @ S
Q = M.T @ M
W = M.T @ S

# Small penalty on the debt required to implement the no-Ponzi scheme
R[0, 0] = R[0, 0] + 1e-9
```

We can now create an instance of `LQ`:

In [4]:
```
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:

In [5]:
```
S - M @ F, (S - M @ F) @ (A - B @ F)
```

Out[5]:
```
(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:

In [6]:
```
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.

```
In [7]: 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()
```

60.5 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.

60.6 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.

In [8]:
```
# 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:

In [9]:
```
lqm.Fs[0]
```

Out[9]:
```
array([[-0.98437712, 19.20516427, -0.8314215 ]])
```

In [10]:
```
lqm.Fs[1]
```

Out[10]:
```
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.

In [11]:
```
T = 2000
x0 = np.array([[1000, 1, 25]])
for i in range(250):
    x, u, w, s = lqm.compute_sequence(x0, ts_length=T)
    plt.plot(list(range(T+1)), x[0, :])
plt.xlabel('Time')
plt.ylabel('Govt Debt')
plt.show()
```

17500
15000
12500
10000
7500
5000
2500
0
Govt Debt
0
250
500
750
1000
1250
1500
1750
2000
Time

Chapter 61

How to Pay for a War: Part 2

61.1 Contents

- An Application of Markov Jump Linear Quadratic Dynamic Programming 61.2
- Two example specifications 61.3
- One- and Two-period Bonds but No Restructuring **??**
- Mapping into an LQ Markov Jump Problem 61.5
- Penalty on Different Issuance Across Maturities 61.6
- A Model with Restructuring 61.7
- Restructuring as a Markov Jump Linear Quadratic Control Problem 61.8

Co-author: Sebastian Graves

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

```
In [1]: !pip install --upgrade quantecon
```

61.2 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 [15], 2003 [16]) for extending his classic 1979 model of tax smoothing.

Barro's 1979 [14] 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 [14] model looks a lot like a consumption-smoothing model.

Our generalizations of his 1979 [14] model will also look like souped-up consumption-smoothing models.

Wanting tractability induced Barro in 1979 [14] 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:

```
In [2]: import quantecon as qe
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

61.3 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.

61.4 One- and Two-period Bonds but No Restructuring

Let T_t denote tax collections, β a discount factor, $b_{t,t+1}$ time $t+1$ goods that the government promises to pay at t, $b_{t,t+2}$ 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) [14], government expenditures are governed by an exogenous stochastic process.

Given initial conditions $b_{-2,0}, b_{-1,0}, z_0, i_0$, where i_0 is the initial Markov state, the government chooses a contingency plan for $\{b_{t,t+1}, b_{t,t+2}, T_t\}_{t=0}^{\infty}$ to maximize.

$$-E_0 \sum_{t=0}^{\infty} \beta^t \left[T_t^2 + c_1(b_{t,t+1} - b_{t,t+2})^2\right]$$

subject to the constraints

$$
\begin{aligned}
T_t &= G_t + b_{t-2,t} + b_{t-1,t} - p_{t,t+2} b_{t,t+2} - p_{t,t+1} b_{t,t+1} \\
G_t &= U_{g,s_t} z_t \\
z_{t+1} &= A_{22,s_t} z_t + C_{2,s_t} w_{t+1} \\
\begin{bmatrix} p_{t,t+1} \\ p_{t,t+2} \\ U_{g,s_t} \\ A_{22,s_t} \\ C_{2,s_t} \end{bmatrix} &\sim \text{functions of Markov state with transition matrix } \Pi
\end{aligned}
$$

Here $w_{t+1} \sim N(0, I)$ and Π_{ij} is the probability that the Markov state moves from state i to state j in one period.

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.

61.5 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

$$M_t = \begin{bmatrix} -p_{t,t+1} & -p_{t,t+2} \end{bmatrix}$$

where $p_{t,t+1}$ is the discount on one period loans in the discrete Markov state at time t and $p_{t,t+2}$ is the discount on two-period loans in the discrete Markov state.

Define

$$S_t = S_{G,t} + S_1$$

Note that in discrete Markov state i

$$T_t = M_t u_t + S_t x_t$$

It follows that

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

or

$$T_t^2 = x_t' R_t x_t + u_t' Q_t u_t + 2u_t' W_t x_t$$

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:

$$T_t^2 + c_1(b_{t,t+1} - b_{t,t+2})^2 = x_t' R_t x_t + u_t' Q_t u_t + 2u_t' W_t x_t + c_1 u_t' Q^c u_t$$

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:

In [3]:
```
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.

61.6 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.

In [4]:
```
# Model parameters
β, Gbar, ρ, σ, c1 = 0.95, 5, 0.8, 1, 0
p1, p2, p3, p4 = β, β**2 - 0.02, β, β**2 + 0.02

# Basic model matrices
A22 = np.array([[1, 0], [Gbar, ρ] ,])
C_2 = np.array([[0], [σ]])
Ug = np.array([[0, 1]])

A1, B1, C1, R1, Q1, W1 = LQ_markov_mapping(A22, C_2, Ug, p1, p2, c1)
A2, B2, C2, R2, Q2, W2 = LQ_markov_mapping(A22, C_2, Ug, p3, p4, c1)

# Small penalties on debt required to implement no-Ponzi scheme
R1[0, 0] = R1[0, 0] + 1e-9
R2[0, 0] = R2[0, 0] + 1e-9

# Construct lists of matrices correspond to each state
As = [A1, A2]
Bs = [B1, B2]
Cs = [C1, C2]
Rs = [R1, R2]
Qs = [Q1, Q2]
Ws = [W1, W2]

Π = np.array([[0.9, 0.1],
              [0.1, 0.9]])

# Construct and solve the model using the LQMarkov class
lqm = qe.LQMarkov(Π, Qs, Rs, As, Bs, Cs=Cs, Ns=Ws, beta=β)
lqm.stationary_values()

# Simulate the model
x0 = np.array([[100, 50, 1, 10]])
x, u, w, t = lqm.compute_sequence(x0, ts_length=300)

# Plot of one and two-period debt issuance
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(u[0, :])
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:

In [5]:
```
# Put small penalty on different issuance across maturities
c1 = 0.01

A1, B1, C1, R1, Q1, W1 = LQ_markov_mapping(A22, C_2, Ug, p1, p2, c1)
A2, B2, C2, R2, Q2, W2 = LQ_markov_mapping(A22, C_2, Ug, p3, p4, c1)

# Small penalties on debt required to implement no-Ponzi scheme
R1[0, 0] = R1[0, 0] + 1e-9
R2[0, 0] = R2[0, 0] + 1e-9

# Construct lists of matrices
As = [A1, A2]
Bs = [B1, B2]
Cs = [C1, C2]
Rs = [R1, R2]
Qs = [Q1, Q2]
Ws = [W1, W2]

# Construct and solve the model using the LQMarkov class
lqm2 = qe.LQMarkov(Π, Qs, Rs, As, Bs, Cs=Cs, Ns=Ws, beta=β)
lqm2.stationary_values()

# Simulate the model
x, u, w, t = lqm2.compute_sequence(x0, ts_length=300)

# Plot of one and two-period debt issuance
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()
```

61.7 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 $S_t = \begin{bmatrix} (\tilde{S}_x + p_t\text{'}S_s\text{'}S_x) & Ug \end{bmatrix}$.

Then

$$T_t = S_t x_t - p_t' u_t$$

Therefore

$$T_t^2 = x_t' R_t x_t + u_t' Q_t u_t + 2u_t' W_t x_t$$

where

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

where to economize on notation we adopt the convention that for the linear state matrices $R_t \equiv R_{s_t}, Q_t \equiv W_{s_t}$ and so on.

We'll 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:

$$T_t^2 + c_2 x_t' S_c' S_c x_t - 2c_2 u_t' S_c x_t + c_2 u_t' u_t$$

Therefore, with the cost term, we must amend our R, Q, W matrices as follows:

$$R_t^c = R_t + c_2 S_c' S_c$$

$$Q_t^c = Q_t + c_2 I$$

$$W_t^c = W_t - c_2 S_c$$

To finish mapping into the Markov jump LQ setup, we need to construct the law of motion for the full state.

This is simpler than in the previous setup, as we now have $\bar{b}_{t+1} = u_t$.

Therefore:

$$x_{t+1} \equiv \begin{bmatrix} \bar{b}_{t+1} \\ z_{t+1} \end{bmatrix} = A_t x_t + B u_t + C_t w_{t+1}$$

where

$$A_t = \begin{bmatrix} 0 & 0 \\ 0 & A_{22,t} \end{bmatrix}, \quad B = \begin{bmatrix} I \\ 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ C_{2,t} \end{bmatrix}$$

This completes the mapping into a Markov jump LQ problem.

61.8 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:

```
In [6]: 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
```

61.8.1 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

In [7]:
```
# 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)
```

In [8]:
```
# Plots of different maturities debt issuance

fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(16, 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')
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()
```

```
In [9]: # Plot share of debt issuance that is short-term

        fig, ax = plt.subplots()
        ax.plot((u[0, :] / (u[0, :] + u[1, :] + u[2, :])))
        ax.set_title('One-period debt issuance share')
        ax.set_xlabel('Time')
        plt.show()
```

Chapter 62

How to Pay for a War: Part 3

62.1 Contents

- Another Application of Markov Jump Linear Quadratic Dynamic Programming 62.2
- Roll-Over Risk 62.3
- A Dead End 62.4
- Better Representation of Roll-Over Risk 62.5

Co-author: Sebastian Graves

In addition to what's in Anaconda, this lecture deploys the quantecon library:

```
In [1]: !pip install --upgrade quantecon
```

62.2 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 [15], 2003 [16]) that extend his classic 1979 [14] model of tax smoothing.

Barro's 1979 [14] 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 [14] 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:

```
In [2]: import quantecon as qe
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

62.3 Roll-Over Risk

Let T_t denote tax collections, β a discount factor, $b_{t,t+1}$ time $t+1$ goods that the government promises to pay at t, G_t government purchases, p_{t+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.

62.4 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.

62.5 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".

In [3]:
```
# Model parameters
β, Gbar, ρ, σ = 0.95, 5, 0.8, 1

# Basic model matrices
A22 = np.array([[1, 0], [Gbar, ρ], ])
C2 = np.array([[0], [σ]])
```

```
Ug = np.array([[0, 1]])

# LQ framework matrices
A_t = np.zeros((1, 3))
A_b = np.hstack((np.zeros((2, 1)), A22))
A = np.vstack((A_t, A_b))

B = np.zeros((3, 1))
B[0, 0] = 1

C = np.vstack((np.zeros((1, 1)), C2))

Sg = np.hstack((np.zeros((1, 1)), Ug))
S1 = np.zeros((1, 3))
S1[0, 0] = 1
S = S1 + Sg

R = S.T @ S

# Large penalty on debt in R2 to prevent borrowing in a bad state
R1 = np.copy(R)
R2 = np.copy(R)
R1[0, 0] = R[0, 0] + 1e-9
R2[0, 0] = R[0, 0] + 1e12

M = np.array([[-β]])
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.

```
In [4]: 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=(16, 4))
ax1.plot(x[0, :])
ax1.set_title('One-period debt issuance')
ax1.set_xlabel('Time')
ax2.plot(tax)
ax2.set_title('Taxation')
ax2.set_xlabel('Time')
plt.show()
```

We can adjust 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 $\beta + 0.02 = 0.97$.

```
In [5]: 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=(16, 4))
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 63

Optimal Taxation in an LQ Economy

63.1 Contents

- Overview 63.2
- The Ramsey Problem 63.3
- Implementation 63.4
- Examples 63.5
- Exercises 63.6
- Solutions 63.7

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

63.2 Overview

In this lecture, we study optimal fiscal policy in a linear quadratic setting.

We modify a model of Robert Lucas and Nancy Stokey [111] 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* [129] 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 poli-

cies.

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 [14].

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:

In [2]:
```
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
```

63.2.1 Model Features

- Linear quadratic (LQ) model
- Representative household
- Stochastic dynamic programming over an infinite horizon
- Distortionary taxation

63.3 The Ramsey Problem

We begin by outlining the key assumptions regarding technology, households and the government sector.

63.3.1 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.

63.3.2 Households

Consider a representative household who chooses a path $\{\ell_t, c_t\}$ for labor and consumption to maximize

$$-\mathbb{E}\frac{1}{2}\sum_{t=0}^{\infty}\beta^t\left[(c_t-b_t)^2+\ell_t^2\right] \tag{1}$$

subject to the budget constraint

$$\mathbb{E}\sum_{t=0}^{\infty}\beta^t p_t^0\left[d_t+(1-\tau_t)\ell_t+s_t-c_t\right]=0 \tag{2}$$

Here

- β is a discount factor in $(0,1)$.
- p_t^0 is a scaled Arrow-Debreu price at time 0 of history contingent goods at time $t+j$.
- b_t is a stochastic preference parameter.
- d_t is an endowment process.
- τ_t is a flat tax rate on labor income.
- s_t is a promised time-t coupon payment on debt issued by the government.

The scaled Arrow-Debreu price p_t^0 is related to the unscaled Arrow-Debreu price as follows.

If we let $\pi_t^0(x^t)$ denote the probability (density) of a history $x^t=[x_t,x_{t-1},\dots,x_0]$ of the state x^t, then the Arrow-Debreu time 0 price of a claim on one unit of consumption at date t, history x^t would be

$$\frac{\beta^t p_t^0}{\pi_t^0(x^t)}$$

Thus, our scaled Arrow-Debreu price is the ordinary Arrow-Debreu price multiplied by the discount factor β^t and divided by an appropriate probability.

The budget constraint (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.

63.3.3 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.

63.3.4 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.

63.3.5 Feasibility

The period-by-period feasibility restriction for this economy is

$$c_t + g_t = d_t + \ell_t \tag{3}$$

A labor-consumption process $\{\ell_t, c_t\}$ is called *feasible* if (3) holds for all t.

63.3.6 Government Budget Constraint

Where p_t^0 is again a scaled Arrow-Debreu price, the time zero government budget constraint is

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t p_t^0 (s_t + g_t - \tau_t \ell_t) = 0 \tag{4}$$

63.3.7 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 (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 (4) in terms of exogenous variables and allocations.
3. Maximize the household's objective function (1) subject to the constraint constructed in step 2 and the feasibility constraint (3).

The solution to this maximization problem pins down all quantities of interest.

63.3.8 Solution

Step one is to obtain the first-conditions for the household's problem, taking taxes and prices as given.

Letting μ be the Lagrange multiplier on (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

$$p_t^0 = \frac{b_t - c_t}{b_0 - c_0} \quad \text{and} \quad \tau_t = 1 - \frac{\ell_t}{b_t - c_t} \tag{5}$$

Substituting (5) into the government's budget constraint (4) yields

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t \left[(b_t - c_t)(s_t + g_t - \ell_t) + \ell_t^2 \right] = 0 \tag{6}$$

The Ramsey problem now amounts to maximizing (1) subject to (6) and (3).

The associated Lagrangian is

$$\mathcal{L} = \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left\{ -\frac{1}{2} \left[(c_t - b_t)^2 + \ell_t^2 \right] + \lambda \left[(b_t - c_t)(\ell_t - s_t - g_t) - \ell_t^2 \right] + \mu_t [d_t + \ell_t - c_t - g_t] \right\} \tag{7}$$

The first-order conditions associated with c_t and ℓ_t are

$$-(c_t - b_t) + \lambda[-\ell_t + (g_t + s_t)] = \mu_t$$

and

$$\ell_t - \lambda[(b_t - c_t) - 2\ell_t] = \mu_t$$

Combining these last two equalities with (3) and working through the algebra, one can show that

$$\ell_t = \bar{\ell}_t - \nu m_t \quad \text{and} \quad c_t = \bar{c}_t - \nu m_t \tag{8}$$

where

- $\nu := \lambda/(1+2\lambda)$
- $\bar{\ell}_t := (b_t - d_t + g_t)/2$
- $\bar{c}_t := (b_t + d_t - g_t)/2$
- $m_t := (b_t - d_t - s_t)/2$

Apart from ν, all of these quantities are expressed in terms of exogenous variables.

To solve for ν, we can use the government's budget constraint again.

The term inside the brackets in (6) is $(b_t - c_t)(s_t + g_t) - (b_t - c_t)\ell_t + \ell_t^2$.

Using (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 (6), we get

$$\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t (b_t - \bar{c}_t)(g_t + s_t)\right\} + (\nu^2 - \nu)\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t 2m_t^2\right\} = 0 \tag{9}$$

Although it might not be clear yet, we are nearly there because:

- The two expectations terms in (9) can be solved for in terms of model primitives.
- This in turn allows us to solve for the Lagrange multiplier ν.
- With ν in hand, we can go back and solve for the allocations via (8).
- Once we have the allocations, prices and the tax system can be derived from (5).

63.3.9 Computing the Quadratic Term

Let's consider how to obtain the term ν in (9).

If we can compute the two expected geometric sums

$$b_0 := \mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t (b_t - \bar{c}_t)(g_t + s_t)\right\} \quad \text{and} \quad a_0 := \mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t 2m_t^2\right\} \tag{10}$$

then the problem reduces to solving

$$b_0 + a_0(\nu^2 - \nu) = 0$$

for ν.

Provided that $4b_0 < a_0$, there is a unique solution $\nu \in (0, 1/2)$, and a unique corresponding $\lambda > 0$.

Let's work out how to compute mathematical expectations in (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 (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

$$q(x_0) = \mathbb{E}\sum_{t=0}^{\infty}\beta^t x_t' H x_t \tag{11}$$

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 $q(x_0) = x_0'Qx_0 + v$, where

- Q is the solution to $Q = H + \beta A'QA$, 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.

63.3.10 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 $\{x^1, \dots, x^N\} \subset \mathbb{R}^k$.

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, $h(x_t) = x_t'Hx_t$.

It is legitimate to pass the expectation through the sum, leading to

$$q(x_0) = \sum_{t=0}^{\infty}\beta^t (P^t h)[j] \tag{12}$$

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) 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.

63.3.11 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

$$B_t := \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j p_{t+j}^t (\tau_{t+j}\ell_{t+j} - g_{t+j}) \tag{13}$$

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 (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

$$B_t = (\tau_t \ell_t - g_t) + \beta E_t p_{t+1}^t B_{t+1} \tag{14}$$

Define

$$R_t^{-1} := \mathbb{E}_t \beta^j p_{t+1}^t \tag{15}$$

R_t is the gross 1-period risk-free rate for loans between t and $t+1$.

63.3.12 A Martingale

We now want to study the following two objects, namely,

$$\pi_{t+1} := B_{t+1} - R_t[B_t - (\tau_t \ell_t - g_t)]$$

and the cumulation of π_t

$$\Pi_t := \sum_{s=0}^{t} \pi_t$$

The term π_{t+1} is the difference between two quantities:

- B_{t+1}, the value of government debt at the start of period $t+1$.
- $R_t[B_t + g_t - \tau_t]$, which is what the government would have owed at the beginning of period $t+1$ if it had simply borrowed at the one-period risk-free rate rather than selling state-contingent securities.

Thus, π_{t+1} is the excess payout on the actual portfolio of state-contingent government debt relative to an alternative portfolio sufficient to finance $B_t + g_t - \tau_t \ell_t$ and consisting entirely of risk-free one-period bonds.

Use expressions (14) and (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

$$\pi_{t+1} = B_{t+1} - \tilde{E}_t B_{t+1} \tag{16}$$

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 (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 [14], 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.

63.4 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

In [3]:
```
# 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
    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 ν = lambda / (1 + 2*lambda).  Here ν is the
# solution to a quadratic equation a(ν**2 - ν) + 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
    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)

    # 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()


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()
```

63.4.1 Comments on the Code

The function `var_quadratic_sum` imported from `quadsums` is for computing the value of (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.

63.5 Examples

Let's look at two examples of usage.

63.5.1 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

In [4]:
```
# == 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.

```
In [5]: gen_fig_2(path)
```

See the original manuscript for comments and interpretation.

63.5.2 The Discrete Case

Our second example adopts a discrete Markov specification for the exogenous process

```
In [6]: # == 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]
        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

```
In [7]: gen_fig_2(path)
```

See the original manuscript for comments and interpretation.

63.6 Exercises

63.6.1 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.

63.7 Solutions

63.7.1 Exercise 1

In [8]:
```
# == 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)
```

```
In [9]: gen_fig_2(path)
```

Chapter 64

Production Smoothing via Inventories

64.1 Contents

- Overview 64.2
- Example 1 64.3
- Inventories Not Useful 64.4
- Inventories Useful but are Hardwired to be Zero Always 64.5
- Example 2 64.6
- Example 3 64.7
- Example 4 64.8
- Example 5 64.9
- Example 6 64.10
- Exercises 64.11

Co-author: Zejin Shi

In addition to what's in Anaconda, this lecture employs the following library:

```
In [1]: !pip install --upgrade quantecon
```

64.2 Overview

This lecture can be viewed as an application of the quantecon lecture.

It formulates a discounted dynamic program for a firm that chooses a production schedule to balance

- minimizing costs of production across time, against
- keeping costs of holding inventories low

In the tradition of a classic book by Holt, Modigliani, Muth, and Simon [**?**], we simplify the firm's problem by formulating it as a linear quadratic discounted dynamic programming problem of the type studied in this quantecon.

Because its costs of production are increasing and quadratic in production, the firm wants to smooth production across time provided that holding inventories is not too costly.

But the firm also prefers to sell out of existing inventories, a preference that we represent by a cost that is quadratic in the difference between sales in a period and the firm's beginning of period inventories.

We compute examples designed to indicate how the firm optimally chooses to smooth production and manage inventories while keeping inventories close to sales.

To introduce components of the model, let

- S_t be sales at time t
- Q_t be production at time t
- I_t be inventories at the beginning of time t
- $\beta \in (0,1)$ be a discount factor
- $c(Q_t) = c_1 Q_t + c_2 Q_t^2$, be a cost of production function, where $c_1 > 0, c_2 > 0$, be an inventory cost function
- $d(I_t, S_t) = d_1 I_t + d_2 (S_t - I_t)^2$, where $d_1 > 0, d_2 > 0$, be a cost-of-holding-inventories function, consisting of two components:
  - a cost $d_1 t$ of carrying inventories, and
  - a cost $d_2 (S_t - I_t)^2$ of having inventories deviate from sales
- $p_t = a_0 - a_1 S_t + v_t$ be an inverse demand function for a firm's product, where $a_0 > 0, a_1 > 0$ and v_t is a demand shock at time t
- $\pi_t = p_t S_t - c(Q_t) - d(I_t, S_t)$ be the firm's profits at time t
- $\sum_{t=0}^{\infty} \beta^t \pi_t$ be the present value of the firm's profits at time 0
- $I_{t+1} = I_t + Q_t - S_t$ be the law of motion of inventories
- $z_{t+1} = A_{22} z_t + C_2 \epsilon_t + 1$ be the law of motion for an exogenous state vector z_t that contains time t information useful for predicting the demand shock v_t
- $v_t = G z_t$ link the demand shock to the information set z_t
- the constant 1 be the first component of z_t

To map our problem into a linear-quadratic discounted dynamic programming problem (also known as an optimal linear regulator), we define the **state** vector at time t as

$$x_t = \begin{bmatrix} I_t \\ z_t \end{bmatrix}$$

and the **control** vector as

$$u_t = \begin{bmatrix} Q_t \\ S_t \end{bmatrix}$$

The law of motion for the state vector x_t is evidently

$$\begin{bmatrix} I_{t+1} \\ z_t \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} I_t \\ z_t \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} Q_t \\ S_t \end{bmatrix} + \begin{bmatrix} 0 \\ C_2 \end{bmatrix} \epsilon_{t+1}$$

or

$$x_{t+1} = A x_t + B u_t + C \epsilon_{t+1}$$

(At this point, we ask that you please forgive us for using Q_t to be the firm's production at time t, while below we use Q as the matrix in the quadratic form $u_t' Q u_t$ that appears in the firm's one-period profit function)

We can express the firm's profit as a function of states and controls as

$$\pi_t = -(x_t' R x_t + u_t' Q u_t + 2 u_t' H x_t)$$

To form the matrices R, Q, H, we note that the firm's profits at time t function can be expressed

$$\begin{aligned}
\pi_t =& p_t S_t - c\left(Q_t\right) - d\left(I_t, S_t\right) \\
=& \left(a_0 - a_1 S_t + v_t\right) S_t - c_1 Q_t - c_2 Q_t^2 - d_1 I_t - d_2 \left(S_t - I_t\right)^2 \\
=& a_0 S_t - a_1 S_t^2 + G z_t S_t - c_1 Q_t - c_2 Q_t^2 - d_1 I_t - d_2 S_t^2 - d_2 I_t^2 + 2 d_2 S_t I_t \\
=& -\left(\underbrace{d_1 I_t + d_2 I_t^2}_{x_t' R x_t} \underbrace{+ a_1 S_t^2 + d_2 S_t^2 + c_2 Q_t^2}_{u_t' Q u_t} \underbrace{- a_0 S_t - G z_t S_t + c_1 Q_t - 2 d_2 S_t I_t}_{2 u_t' H x_t}\right) \\
=& -\left(\begin{bmatrix} I_t & z_t' \end{bmatrix} \underbrace{\begin{bmatrix} d_2 & \frac{d_1}{2} S_c \\ \frac{d_1}{2} S_c' & 0 \end{bmatrix}}_{\equiv R} \begin{bmatrix} I_t \\ z_t \end{bmatrix} + \begin{bmatrix} Q_t & S_t \end{bmatrix} \underbrace{\begin{bmatrix} c_2 & 0 \\ 0 & a_1 + d_2 \end{bmatrix}}_{\equiv Q} \begin{bmatrix} Q_t \\ S_t \end{bmatrix} + 2 \begin{bmatrix} Q_t & S_t \end{bmatrix} \underbrace{\begin{bmatrix} 0 & \frac{c_1}{2} \\ -d_2 & -\frac{a_0}{2} S \end{bmatrix}}_{\equiv N}\right.
\end{aligned} \tag{64.1}$$

where $S_c = [1, 0]$.

Remark on notation: The notation for cross product term in the QuantEcon library is N instead of H.

The firms' optimum decision rule takes the form

$$u_t = -F x_t$$

and the evolution of the state under the optimal decision rule is

$$x_{t+1} = (A - BF) x_t + C \epsilon_{t+1}$$

Here is code for computing an optimal decision rule and for analyzing its consequences.

In [2]:
```
import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
%matplotlib inline
```

In [3]:
```
class smoothing_example:
    """
    Class for constructing, solving, and plotting results for
    inventories and sales smoothing problem.
    """

    def __init__(self,
                 β=0.96,            # Discount factor
                 c1=1,              # Cost-of-production
                 c2=1,
                 d1=1,              # Cost-of-holding inventories
                 d2=1,
```

```
                 a0=10,            # Inverse demand function
                 a1=1,
                 A22=[[1,   0],    # z process
                      [1, 0.9]],
                 C2=[[0], [1]],
                 G=[0, 1]):

        self.β = β
        self.c1, self.c2 = c1, c2
        self.d1, self.d2 = d1, d2
        self.a0, self.a1 = a0, a1
        self.A22 = np.atleast_2d(A22)
        self.C2 = np.atleast_2d(C2)
        self.G = np.atleast_2d(G)

        # Dimensions
        k, j = self.C2.shape        # Dimensions for randomness part
        n = k + 1                   # Number of states
        m = 2                       # Number of controls

        Sc = np.zeros(k)
        Sc[0] = 1

        # Construct matrices of transition law
        A = np.zeros((n, n))
        A[0, 0] = 1
        A[1:, 1:] = A22

        B = np.zeros((n, m))
        B[0, :] = 1, -1

        C = np.zeros((n, j))
        C[1:, :] = C2

        self.A, self.B, self.C = A, B, C

        # Construct matrices of one period profit function
        R = np.zeros((n, n))
        R[0, 0] = d2
        R[1:, 0] = d1 / 2 * Sc
        R[0, 1:] = d1 / 2 * Sc

        Q = np.zeros((m, m))
        Q[0, 0] = c2
        Q[1, 1] = a1 + d2

        N = np.zeros((m, n))
        N[1, 0] = - d2
        N[0, 1:] = c1 / 2 * Sc
        N[1, 1:] = - a0 / 2 * Sc - self.G / 2

        self.R, self.Q, self.N = R, Q, N

        # Construct LQ instance
        self.LQ = qe.LQ(Q, R, A, B, C, N, beta=β)
        self.LQ.stationary_values()

    def simulate(self, x0, T=100):
```

```
        c1, c2 = self.c1, self.c2
        d1, d2 = self.d1, self.d2
        a0, a1 = self.a0, self.a1
        G = self.G

        x_path, u_path, w_path = self.LQ.compute_sequence(x0, ts_length=T)

        I_path = x_path[0, :-1]
        z_path = x_path[1:, :-1]
        □_path = (G @ z_path)[0, :]

        Q_path = u_path[0, :]
        S_path = u_path[1, :]

        revenue = (a0 - a1 * S_path + □_path) * S_path
        cost_production = c1 * Q_path + c2 * Q_path ** 2
        cost_inventories = d1 * I_path + d2 * (S_path - I_path) ** 2

        Q_no_inventory = (a0 + □_path - c1) / (2 * (a1 + c2))
        Q_hardwired = (a0 + □_path - c1) / (2 * (a1 + c2 + d2))

        fig, ax = plt.subplots(2, 2, figsize=(15, 10))

        ax[0, 0].plot(range(T), I_path, label="inventories")
        ax[0, 0].plot(range(T), S_path, label="sales")
        ax[0, 0].plot(range(T), Q_path, label="production")
        ax[0, 0].legend(loc=1)
        ax[0, 0].set_title("inventories, sales, and production")

        ax[0, 1].plot(range(T), (Q_path - S_path), color='b')
        ax[0, 1].set_ylabel("change in inventories", color='b')
        span = max(abs(Q_path - S_path))
        ax[0, 1].set_ylim(0-span*1.1, 0+span*1.1)
        ax[0, 1].set_title("demand shock and change in inventories")

        ax1_ = ax[0, 1].twinx()
        ax1_.plot(range(T), □_path, color='r')
        ax1_.set_ylabel("demand shock", color='r')
        span = max(abs(□_path))
        ax1_.set_ylim(0-span*1.1, 0+span*1.1)

        ax1_.plot([0, T], [0, 0], '--', color='k')

        ax[1, 0].plot(range(T), revenue, label="revenue")
        ax[1, 0].plot(range(T), cost_production, label="cost_production")
        ax[1, 0].plot(range(T), cost_inventories, label="cost_inventories")
        ax[1, 0].legend(loc=1)
        ax[1, 0].set_title("profits decomposition")

        ax[1, 1].plot(range(T), Q_path, label="production")
        ax[1, 1].plot(range(T), Q_hardwired, label='production when  $I_t$ \
            forced to be zero')
        ax[1, 1].plot(range(T), Q_no_inventory, label='production when \
            inventories not useful')
        ax[1, 1].legend(loc=1)
        ax[1, 1].set_title('three production concepts')
```

```
plt.show()
```

Notice that the above code sets parameters at the following default values

- discount factor β=0.96,
- inverse demand function: $a0 = 10, a1 = 1$
- cost of production $c1 = 1, c2 = 1$
- costs of holding inventories $d1 = 1, d2 = 1$

In the examples below, we alter some or all of these parameter values.

64.3 Example 1

In this example, the demand shock follows AR(1) process:

$$\nu_t = \alpha + \rho \nu_{t-1} + \epsilon_t,$$

which implies

$$z_{t+1} = \begin{bmatrix} 1 \\ v_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \alpha & \rho \end{bmatrix} \underbrace{\begin{bmatrix} 1 \\ v_t \end{bmatrix}}_{z_t} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \epsilon_{t+1}.$$

We set $\alpha = 1$ and $\rho = 0.9$, their default values.

We'll calculate and display outcomes, then discuss them below the pertinent figures.

```
In [4]: ex1 = smoothing_example()

        x0 = [0, 1, 0]
        ex1.simulate(x0)
```

The figures above illustrate various features of an optimal production plan.

Starting from zero inventories, the firm builds up a stock of inventories and uses them to smooth costly production in the face of demand shocks.

Optimal decisions evidently respond to demand shocks.

Inventories are always less than sales, so some sales come from current production, a consequence of the cost, $d_1 I_t$ of holding inventories.

The lower right panel shows differences between optimal production and two alternative production concepts that come from altering the firm's cost structure – i.e., its technology.

These two concepts correspond to these distinct altered firm problems.

- a setting in which inventories are not needed
- a setting in which they are needed but we arbitrarily prevent the firm from holding inventories by forcing it to set $I_t = 0$ always

We use these two alternative production concepts in order to shed light on the baseline model.

64.4 Inventories Not Useful

Let's turn first to the setting in which inventories aren't needed.

In this problem, the firm forms an output plan that maximizes the expected value of

$$\sum_{t=0}^{\infty} \beta^t \{p_t Q_t - C(Q_t)\}$$

It turns out that the optimal plan for Q_t for this problem also solves a sequence of static problems $\max_{Q_t} \{p_t Q_t - c(Q_t)\}$.

When inventories aren't required or used, sales always equal production.

This simplifies the problem and the optimal no-inventory production maximizes the expected value of

$$\sum_{t=0}^{\infty} \beta^t \left\{p_t Q_t - C\left(Q_t\right)\right\}.$$

The optimum decision rule is

$$Q_t^{ni} = \frac{a_0 + \nu_t - c_1}{c_2 + a_1}.$$

64.5 Inventories Useful but are Hardwired to be Zero Always

Next, we turn to a distinct problem in which inventories are useful – meaning that there are costs of $d_2(I_t - S_t)^2$ associated with having sales not equal to inventories – but we arbitrarily impose on the firm the costly restriction that it never hold inventories.

Here the firm's maximization problem is

$$\max_{\{I_t, Q_t, S_t\}} \sum_{t=0}^{\infty} \beta^t \left\{ p_t S_t - C\left(Q_t\right) - d\left(I_t, S_t\right) \right\}$$

subject to the restrictions that $I_t = 0$ for all t and that $I_{t+1} = I_t + Q_t - S_t$.

The restriction that $I_t = 0$ implies that $Q_t = S_t$ and that the maximization problem reduces to

$$\max_{Q_t} \sum_{t=0}^{\infty} \beta^t \left\{ p_t Q_t - C\left(Q_t\right) - d\left(0, Q_t\right) \right\}$$

Here the optimal production plan is

$$Q_t^h = \frac{a_0 + \nu_t - c_1}{c_2 + a_1 + d_2}.$$

We introduce this I_t **is hardwired to zero** specification in order to shed light on the role that inventories play by comparing outcomes with those under our two other versions of the problem.

The bottom right panel displays an production path for the original problem that we are interested in (the blue line) as well with an optimal production path for the model in which inventories are not useful (the green path) and also for the model in which, although inventories are useful, they are hardwired to zero and the firm pays cost $d(0, Q_t)$ for not setting sales $S_t = Q_t$ equal to zero (the orange line).

Notice that it is typically optimal for the firm to produce more when inventories aren't useful. Here there is no requirement to sell out of inventories and no costs from having sales deviate from inventories.

But "typical" does not mean "always".

Thus, if we look closely, we notice that for small t, the green "production when inventories aren't useful" line in the lower right panel is below optimal production in the original model.

High optimal production in the original model early on occurs because the firm wants to accumulate inventories quickly in order to acquire high inventories for use in later periods.

But how the green line compares to the blue line early on depends on the evolution of the demand shock, as we will see in a deterministically seasonal demand shock example to be analyzed below.

In that example, the original firm optimally accumulates inventories slowly because the next positive demand shock is in the distant future.

To make the green-blue model production comparison easier to see, let's confine the graphs to the first 10 periods:

```
In [5]: ex1.simulate(x0, T=10)
```

64.6 Example 2

Next, we shut down randomness in demand and assume that the demand shock ν_t follows a deterministic path:

$$\nu_t = \alpha + \rho \nu_{t-1}$$

Again, we'll compute and display outcomes in some figures

```
In [6]: ex2 = smoothing_example(C2=[[0], [0]])

        x0 = [0, 1, 0]
        ex2.simulate(x0)
```

64.7 Example 3

Now we'll put randomness back into the demand shock process and also assume that there are zero costs of holding inventories.

In particular, we'll look at a situation in which $d_1 = 0$ but $d_2 > 0$.

Now it becomes optimal to set sales approximately equal to inventories and to use inventories to smooth production quite well, as the following figures confirm

```
In [7]: ex3 = smoothing_example(d1=0)

        x0 = [0, 1, 0]
        ex3.simulate(x0)
```

64.8 Example 4

To bring out some features of the optimal policy that are related to some technical issues in linear control theory, we'll now temporarily assume that it is costless to hold inventories.

When we completely shut down the cost of holding inventories by setting $d_1 = 0$ and $d_2 = 0$, something absurd happens (because the Bellman equation is opportunistic and very smart).

(Technically, we have set parameters that end up violating conditions needed to assure **stability** of the optimally controlled state.)

The firm finds it optimal to set $Q_t \equiv Q^* = \frac{-c_1}{2c_2}$, an output level that sets the costs of production to zero (when $c_1 > 0$, as it is with our default settings, then it is optimal to set production negative, whatever that means!).

Recall the law of motion for inventories

$$I_{t+1} = I_t + Q_t - S_t$$

So when $d_1 = d_2 = 0$ so that the firm finds it optimal to set $Q_t = \frac{-c_1}{2c_2}$ for all t, then

$$I_{t+1} - I_t = \frac{-c_1}{2c_2} - S_t < 0$$

for almost all values of S_t under our default parameters that keep demand positive almost all of the time.

The dynamic program instructs the firm to set production costs to zero and to **run a Ponzi scheme** by running inventories down forever.

(We can interpret this as the firm somehow **going short in** or **borrowing** inventories)

The following figures confirm that inventories head south without limit

```
In [8]: ex4 = smoothing_example(d1=0, d2=0)

        x0 = [0, 1, 0]
        ex4.simulate(x0)
```

Let's shorten the time span displayed in order to highlight what is going on.

We'll set the horizon $T = 30$ with the following code

```
In [9]: # shorter period
        ex4.simulate(x0, T=30)
```

64.9 Example 5

Now we'll assume that the demand shock that follows a linear time trend

$$v_t = b + at, a > 0, b > 0$$

To represent this, we set $C_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and

$$A_{22} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, x_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, G = \begin{bmatrix} b & a \end{bmatrix}$$

```
In [10]: # Set parameters
         a = 0.5
         b = 3.
```

```
In [11]: ex5 = smoothing_example(A22=[[1, 0], [1, 1]], C2=[[0], [0]], G=[b, a])

         x0 = [0, 1, 0] # set the initial inventory as 0
         ex5.simulate(x0, T=10)
```

64.10 Example 6

Now we'll assume a deterministically seasonal demand shock.

To represent this we'll set

$$A_{22} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}, C_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, G' = \begin{bmatrix} b \\ a \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

where $a > 0, b > 0$ and

$$x_0 = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

```
In [12]: ex5 = smoothing_example(A22=[[1, 0, 0, 0, 0],
                                      [0, 0, 0, 0, 1],
                                      [0, 1, 0, 0, 0],
                                      [0, 0, 1, 0, 0],
                                      [0, 0, 0, 1, 0]],
                                 C2=[[0], [0], [0], [0], [0]],
                                 G=[b, a, 0, 0, 0])
```

```
x00 = [0, 1, 0, 1, 0, 0] # Set the initial inventory as 0
ex5.simulate(x00, T=20)
```

Now we'll generate some more examples that differ simply from the initial **season** of the year in which we begin the demand shock

```
In [13]: x01 = [0, 1, 1, 0, 0, 0]
         ex5.simulate(x01, T=20)
```

```
In [14]: x02 = [0, 1, 0, 0, 1, 0]
         ex5.simulate(x02, T=20)
```

```
In [15]: x03 = [0, 1, 0, 0, 0, 1]
         ex5.simulate(x03, T=20)
```

64.11 Exercises

Please try to analyze some inventory sales smoothing problems using the `smoothing_example` class.

64.11.1 Exercise 1

Assume that the demand shock follows AR(2) process below:

$$\nu_t = \alpha + \rho_1 \nu_{t-1} + \rho_2 \nu_{t-2} + \epsilon_t.$$

You need to construct $A22$, C, and G matrices properly and then to input them as the keyword arguments of `smoothing_example` class. Simulate paths starting from the initial condition $x_0 = [0, 1, 0, 0]'$.

After this, try to construct a very similar `smoothing_example` with the same demand shock process but exclude the randomness ϵ_t. Compute the stationary states $\bar{x}$ by simulating for a long period. Then try to add shocks with different magnitude to $\bar{\nu}_t$ and simulate paths. You should see how firms respond differently by staring at the production plans.

64.11.2 Exercise 2

Change parameters of $C(Q_t)$ and $d(I_t, S_t)$.

1. Make production more costly, by setting $c_2 = 5$.

2. Increase the cost of having inventories deviate from sales, by setting $d_2 = 5$.

64.11.3 Solution 1

In [16]:
```
# set parameters
α = 1
ρ1 = 1.2
ρ2 = -.3
```

In [17]:
```
# construct matrices
A22 =[[1,  0,  0],
          [1, ρ1, ρ2],
          [0,  1, 0]]
C2 = [[0], [1], [0]]
G = [0, 1, 0]
```

In [18]:
```
ex1 = smoothing_example(A22=A22, C2=C2, G=G)

x0 = [0, 1, 0, 0] # initial condition
ex1.simulate(x0)
```

```
In [19]: # now silence the noise
         ex1_no_noise = smoothing_example(A22=A22, C2=[[0], [0], [0]], G=G)

         # initial condition
         x0 = [0, 1, 0, 0]

         # compute stationary states
         x_bar = ex1_no_noise.LQ.compute_sequence(x0, ts_length=250)[0][:, -1]
         x_bar
```

```
Out[19]: array([ 3.69387755,  1.        , 10.        , 10.        ])
```

In the following, we add small and large shocks to $\bar{\nu}_t$ and compare how firm responds differently in quantity. As the shock is not very persistent under the parameterization we are using, we focus on a short period response.

```
In [20]: T = 40
```

```
In [21]: # small shock
         x_bar1 = x_bar.copy()
         x_bar1[2] += 2
         ex1_no_noise.simulate(x_bar1, T=T)
```

```
In [22]: # large shock
         x_bar1 = x_bar.copy()
         x_bar1[2] += 10
         ex1_no_noise.simulate(x_bar1, T=T)
```

64.11.4 Solution 2

```
In [23]: x0 = [0, 1, 0]
```

```
In [24]: smoothing_example(c2=5).simulate(x0)
```

```
In [25]: smoothing_example(d2=5).simulate(x0)
```

Part VIII

Data and Empirics

Chapter 65

Pandas for Panel Data

65.1 Contents

- Overview 65.2
- Slicing and Reshaping Data 65.3
- Merging Dataframes and Filling NaNs 65.4
- Grouping and Summarizing Data 65.5
- Final Remarks 65.6
- Exercises 65.7
- Solutions 65.8

65.2 Overview

In an earlier lecture on pandas, we looked at working with simple data sets.

Econometricians often need to work with more complex data sets, such as panels.

Common tasks include

- Importing data, cleaning it and reshaping it across several axes.
- Selecting a time series or cross-section from a panel.
- Grouping and summarizing data.

`pandas` (derived from 'panel' and 'data') contains powerful and easy-to-use tools for solving exactly these kinds of problems.

In what follows, we will use a panel data set of real minimum wages from the OECD to create:

- summary statistics over multiple dimensions of our data
- a time series of the average minimum wage of countries in the dataset
- kernel density estimates of wages by continent

We will begin by reading in our long format panel data from a CSV file and reshaping the resulting `DataFrame` with `pivot_table` to build a `MultiIndex`.

Additional detail will be added to our `DataFrame` using pandas' `merge` function, and data will be summarized with the `groupby` function.

Most of this lecture was created by Natasha Watkins.

65.3 Slicing and Reshaping Data

We will read in a dataset from the OECD of real minimum wages in 32 countries and assign it to `realwage`.

The dataset can be accessed with the following link:

```
In [1]: url1 = 'https://raw.githubusercontent.com/QuantEcon/lecture-source-
        py/master/source/_static/lecture_specific/pandas_panel/realwage.csv'
```

```
In [2]: import pandas as pd

        # Display 6 columns for viewing purposes
        pd.set_option('display.max_columns', 6)

        # Reduce decimal points to 2
        pd.options.display.float_format = '{:,.2f}'.format

        realwage = pd.read_csv(url1)
```

Let's have a look at what we've got to work with

```
In [3]: realwage.head()  # Show first 5 rows
```

```
Out[3]:    Unnamed: 0       Time  Country                                   Series  \
        0           0  2006-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
        1           1  2007-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
        2           2  2008-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
        3           3  2009-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
        4           4  2010-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs

          Pay period     value
        0     Annual 17,132.44
        1     Annual 18,100.92
        2     Annual 17,747.41
        3     Annual 18,580.14
        4     Annual 18,755.83
```

The data is currently in long format, which is difficult to analyze when there are several dimensions to the data.

We will use `pivot_table` to create a wide format panel, with a `MultiIndex` to handle higher dimensional data.

`pivot_table` arguments should specify the data (values), the index, and the columns we want in our resulting dataframe.

By passing a list in columns, we can create a `MultiIndex` in our column axis

```
In [4]: realwage = realwage.pivot_table(values='value',
                                        index='Time',
                                       columns=['Country', 'Series', 'Pay period'])
        realwage.head()
```

Out[4]:
```
Country                                     Australia         \
Series     In 2015 constant prices at 2015 USD PPPs
Pay period                                     Annual Hourly
Time
2006-01-01                                  20,410.65  10.33
2007-01-01                                  21,087.57  10.67
2008-01-01                                  20,718.24  10.48
2009-01-01                                  20,984.77  10.62
2010-01-01                                  20,879.33  10.57

Country                                                          …  \
Series     In 2015 constant prices at 2015 USD exchange rates  …
Pay period                                                Annual  …
Time                                                              …
2006-01-01                                             23,826.64  …
2007-01-01                                             24,616.84  …
2008-01-01                                             24,185.70  …
2009-01-01                                             24,496.84  …
2010-01-01                                             24,373.76  …

Country                                 United States  \
Series     In 2015 constant prices at 2015 USD PPPs
Pay period                                     Hourly
Time
2006-01-01                                       6.05
2007-01-01                                       6.24
2008-01-01                                       6.78
2009-01-01                                       7.58
2010-01-01                                       7.88

Country
Series     In 2015 constant prices at 2015 USD exchange rates
Pay period                                                Annual Hourly
Time
2006-01-01                                             12,594.40   6.05
2007-01-01                                             12,974.40   6.24
2008-01-01                                             14,097.56   6.78
2009-01-01                                             15,756.42   7.58
2010-01-01                                             16,391.31   7.88

[5 rows x 128 columns]
```

To more easily filter our time series data, later on, we will convert the index into a `DateTimeIndex`

In [5]:
```
realwage.index = pd.to_datetime(realwage.index)
type(realwage.index)
```

Out[5]:
```
pandas.core.indexes.datetimes.DatetimeIndex
```

The columns contain multiple levels of indexing, known as a `MultiIndex`, with levels being ordered hierarchically (Country > Series > Pay period).

A `MultiIndex` is the simplest and most flexible way to manage panel data in pandas

In [6]:
```
type(realwage.columns)
```

Out[6]: pandas.core.indexes.multi.MultiIndex

In [7]:
```
realwage.columns.names
```

Out[7]: FrozenList(['Country', 'Series', 'Pay period'])

Like before, we can select the country (the top level of our `MultiIndex`)

In [8]:
```
realwage['United States'].head()
```

Out[8]:
```
Series     In 2015 constant prices at 2015 USD PPPs         \
Pay period                                    Annual Hourly
Time
2006-01-01                                12,594.40   6.05
2007-01-01                                12,974.40   6.24
2008-01-01                                14,097.56   6.78
2009-01-01                                15,756.42   7.58
2010-01-01                                16,391.31   7.88

Series     In 2015 constant prices at 2015 USD exchange rates
Pay period                                            Annual Hourly
Time
2006-01-01                                          12,594.40   6.05
2007-01-01                                          12,974.40   6.24
2008-01-01                                          14,097.56   6.78
2009-01-01                                          15,756.42   7.58
2010-01-01                                          16,391.31   7.88
```

Stacking and unstacking levels of the `MultiIndex` will be used throughout this lecture to reshape our dataframe into a format we need.

`.stack()` rotates the lowest level of the column `MultiIndex` to the row index (`.unstack()` works in the opposite direction - try it out)

In [9]:
```
realwage.stack().head()
```

Out[9]:
```
Country                                                Australia  \
Series                 In 2015 constant prices at 2015 USD PPPs
Time       Pay period
2006-01-01 Annual                                      20,410.65
           Hourly                                          10.33
2007-01-01 Annual                                      21,087.57
           Hourly                                          10.67
2008-01-01 Annual                                      20,718.24

Country                                                                  \
Series                 In 2015 constant prices at 2015 USD exchange rates
Time       Pay period
2006-01-01 Annual                                                 23,826.64
           Hourly                                                     12.06
2007-01-01 Annual                                                 24,616.84
           Hourly                                                     12.46
2008-01-01 Annual                                                 24,185.70
```

```
Country                                                   Belgium  …  \
Series                In 2015 constant prices at 2015 USD PPPs  …
Time       Pay period                                             …
2006-01-01 Annual                                       21,042.28  …
           Hourly                                           10.09  …
2007-01-01 Annual                                       21,310.05  …
           Hourly                                           10.22  …
2008-01-01 Annual                                       21,416.96  …

Country                                                     United Kingdom  \
Series                In 2015 constant prices at 2015 USD exchange rates
Time       Pay period
2006-01-01 Annual                                                20,376.32
           Hourly                                                     9.81
2007-01-01 Annual                                                20,954.13
           Hourly                                                    10.07
2008-01-01 Annual                                                20,902.87

Country                                             United States  \
Series                In 2015 constant prices at 2015 USD PPPs
Time       Pay period
2006-01-01 Annual                                       12,594.40
           Hourly                                            6.05
2007-01-01 Annual                                       12,974.40
           Hourly                                            6.24
2008-01-01 Annual                                       14,097.56

Country
Series                In 2015 constant prices at 2015 USD exchange rates
Time       Pay period
2006-01-01 Annual                                                12,594.40
           Hourly                                                     6.05
2007-01-01 Annual                                                12,974.40
           Hourly                                                     6.24
2008-01-01 Annual                                                14,097.56

[5 rows x 64 columns]
```

We can also pass in an argument to select the level we would like to stack

```
In [10]: realwage.stack(level='Country').head()
```

```
Out[10]: Series                In 2015 constant prices at 2015 USD PPPs         \
         Pay period                                              Annual Hourly
         Time       Country
         2006-01-01 Australia                                 20,410.65  10.33
                    Belgium                                   21,042.28  10.09
                    Brazil                                     3,310.51   1.41
                    Canada                                    13,649.69   6.56
                    Chile                                      5,201.65   2.22

         Series                In 2015 constant prices at 2015 USD exchange rates
         Pay period                                                     Annual Hourly
         Time       Country
         2006-01-01 Australia                                        23,826.64  12.06
                    Belgium                                          20,228.74   9.70
                    Brazil                                            2,032.87   0.87
```

```
Canada                                                  14,335.12   6.89
Chile                                                    3,333.76   1.42
```

Using a `DatetimeIndex` makes it easy to select a particular time period.

Selecting one year and stacking the two lower levels of the `MultiIndex` creates a cross-section of our panel data

```
In [11]: realwage['2015'].stack(level=(1, 2)).transpose().head()
```

```
Out[11]: Time                                      2015-01-01         \
         Series     In 2015 constant prices at 2015 USD PPPs
         Pay period                                    Annual Hourly
         Country
         Australia                                   21,715.53  10.99
         Belgium                                     21,588.12  10.35
         Brazil                                       4,628.63   2.00
         Canada                                      16,536.83   7.95
         Chile                                        6,633.56   2.80

         Time
         Series     In 2015 constant prices at 2015 USD exchange rates
         Pay period                                              Annual Hourly
         Country
         Australia                                             25,349.90  12.83
         Belgium                                               20,753.48   9.95
         Brazil                                                 2,842.28   1.21
         Canada                                                17,367.24   8.35
         Chile                                                  4,251.49   1.81
```

For the rest of lecture, we will work with a dataframe of the hourly real minimum wages across countries and time, measured in 2015 US dollars.

To create our filtered dataframe (`realwage_f`), we can use the `xs` method to select values at lower levels in the multiindex, while keeping the higher levels (countries in this case)

```
In [12]: realwage_f = realwage.xs(('Hourly', 'In 2015 constant prices at 2015 USD
 ↪exchange
         rates'),
                                 level=('Pay period', 'Series'), axis=1)
         realwage_f.head()
```

```
Out[12]: Country     Australia  Belgium  Brazil  …  Turkey  United Kingdom  \
         Time                                    …
         2006-01-01      12.06     9.70    0.87  …    2.27            9.81
         2007-01-01      12.46     9.82    0.92  …    2.26           10.07
         2008-01-01      12.24     9.87    0.96  …    2.22           10.04
         2009-01-01      12.40    10.21    1.03  …    2.28           10.15
         2010-01-01      12.34    10.05    1.08  …    2.30            9.96

         Country     United States
         Time
         2006-01-01           6.05
         2007-01-01           6.24
         2008-01-01           6.78
```

```
2009-01-01           7.58
2010-01-01           7.88

[5 rows x 32 columns]
```

65.4 Merging Dataframes and Filling NaNs

Similar to relational databases like SQL, pandas has built in methods to merge datasets together.

Using country information from WorldData.info, we'll add the continent of each country to `realwage_f` with the `merge` function.

The dataset can be accessed with the following link:

```
In [13]: url2 = 'https://raw.githubusercontent.com/QuantEcon/lecture-source-
         py/master/source/_static/lecture_specific/pandas_panel/countries.csv'
```

```
In [14]: worlddata = pd.read_csv(url2, sep=';')
         worlddata.head()
```

```
Out[14]:     Country (en) Country (de)        Country (local)  … Deathrate  \
         0    Afghanistan  Afghanistan  Afganistan/Afqanestan  …     13.70
         1          Egypt      Ägypten                   Misr  …      4.70
         2  Åland Islands  Ålandinseln                  Åland  …      0.00
         3        Albania     Albanien              Shqipëria  …      6.70
         4        Algeria     Algerien     Al-Jaza'ir/Algérie  …      4.30

           Life expectancy                                                Url
         0           51.30  https://www.laenderdaten.info/Asien/Afghanista…
         1           72.70  https://www.laenderdaten.info/Afrika/Aegypten/…
         2            0.00  https://www.laenderdaten.info/Europa/Aland/ind…
         3           78.30  https://www.laenderdaten.info/Europa/Albanien/…
         4           76.80  https://www.laenderdaten.info/Afrika/Algerien/…

         [5 rows x 17 columns]
```

First, we'll select just the country and continent variables from `worlddata` and rename the column to 'Country'

```
In [15]: worlddata = worlddata[['Country (en)', 'Continent']]
         worlddata = worlddata.rename(columns={'Country (en)': 'Country'})
         worlddata.head()
```

```
Out[15]:          Country Continent
         0    Afghanistan      Asia
         1          Egypt    Africa
         2  Åland Islands    Europe
         3        Albania    Europe
         4        Algeria    Africa
```

We want to merge our new dataframe, `worlddata`, with `realwage_f`.

The pandas `merge` function allows dataframes to be joined together by rows.

Our dataframes will be merged using country names, requiring us to use the transpose of `realwage_f` so that rows correspond to country names in both dataframes

```
In [16]: realwage_f.transpose().head()
```

```
Out[16]: Time       2006-01-01  2007-01-01  2008-01-01  …  2014-01-01  2015-01-01  \
         Country                                          …
         Australia        12.06       12.46       12.24  …       12.67       12.83
         Belgium           9.70        9.82        9.87  …       10.01        9.95
         Brazil            0.87        0.92        0.96  …        1.21        1.21
         Canada            6.89        6.96        7.24  …        8.22        8.35
         Chile             1.42        1.45        1.44  …        1.76        1.81

         Time       2016-01-01
         Country
         Australia        12.98
         Belgium           9.76
         Brazil            1.24
         Canada            8.48
         Chile             1.91

         [5 rows x 11 columns]
```

We can use either left, right, inner, or outer join to merge our datasets:

- left join includes only countries from the left dataset
- right join includes only countries from the right dataset
- outer join includes countries that are in either the left and right datasets
- inner join includes only countries common to both the left and right datasets

By default, `merge` will use an inner join.

Here we will pass `how='left'` to keep all countries in `realwage_f`, but discard countries in `worlddata` that do not have a corresponding data entry `realwage_f`.

This is illustrated by the red shading in the following diagram

We will also need to specify where the country name is located in each dataframe, which will be the `key` that is used to merge the dataframes 'on'.

Our 'left' dataframe (`realwage_f.transpose()`) contains countries in the index, so we set `left_index=True`.

Our 'right' dataframe (`worlddata`) contains countries in the 'Country' column, so we set `right_on='Country'`

```
In [17]: merged = pd.merge(realwage_f.transpose(), worlddata,
                           how='left', left_index=True, right_on='Country')
         merged.head()
```

```
Out[17]:      2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00  …  \
         17                 12.06                12.46                12.24  …
         23                  9.70                 9.82                 9.87  …
         32                  0.87                 0.92                 0.96  …
         100                 6.89                 6.96                 7.24  …
         38                  1.42                 1.45                 1.44  …

              2016-01-01 00:00:00    Country      Continent
         17                 12.98  Australia      Australia
         23                  9.76    Belgium         Europe
         32                  1.24     Brazil  South America
         100                 8.48     Canada  North America
         38                  1.91      Chile  South America

         [5 rows x 13 columns]
```

Countries that appeared in `realwage_f` but not in `worlddata` will have `NaN` in the Continent column.

To check whether this has occurred, we can use `.isnull()` on the continent column and filter the merged dataframe

```
In [18]: merged[merged['Continent'].isnull()]
```

```
Out[18]:      2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00  …  \
         247                 3.42                 3.74                 3.87  …
         247                 0.23                 0.45                 0.39  …
         247                 1.50                 1.64                 1.71  …

              2016-01-01 00:00:00             Country  Continent
         247                 5.28               Korea        NaN
         247                 0.55  Russian Federation        NaN
         247                 2.08     Slovak Republic        NaN

         [3 rows x 13 columns]
```

We have three missing values!

One option to deal with NaN values is to create a dictionary containing these countries and their respective continents.

`.map()` will match countries in `merged['Country']` with their continent from the dictionary.

Notice how countries not in our dictionary are mapped with `NaN`

```
In [19]: missing_continents = {'Korea': 'Asia',
                               'Russian Federation': 'Europe',
                               'Slovak Republic': 'Europe'}

         merged['Country'].map(missing_continents)
```

```
Out[19]: 17        NaN
         23        NaN
         32        NaN
         100       NaN
         38        NaN
         108       NaN
         41        NaN
         225       NaN
         53        NaN
         58        NaN
         45        NaN
         68        NaN
         233       NaN
         86        NaN
         88        NaN
         91        NaN
         247      Asia
         117       NaN
         122       NaN
         123       NaN
         138       NaN
         153       NaN
         151       NaN
         174       NaN
         175       NaN
         247    Europe
         247    Europe
         198       NaN
         200       NaN
         227       NaN
```

```
241       NaN
240       NaN
Name: Country, dtype: object
```

We don't want to overwrite the entire series with this mapping.

`.fillna()` only fills in `NaN` values in `merged['Continent']` with the mapping, while leaving other values in the column unchanged

```
In [20]: merged['Continent'] =
         merged['Continent'].fillna(merged['Country'].map(missing_continents))

         # Check for whether continents were correctly mapped

         merged[merged['Country'] == 'Korea']
```

```
Out[20]:      2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00  …  \
         247                 3.42                 3.74                 3.87  …

              2016-01-01 00:00:00  Country  Continent
         247                 5.28    Korea       Asia

         [1 rows x 13 columns]
```

We will also combine the Americas into a single continent - this will make our visualization nicer later on.

To do this, we will use `.replace()` and loop through a list of the continent values we want to replace

```
In [21]: replace = ['Central America', 'North America', 'South America']

         for country in replace:
             merged['Continent'].replace(to_replace=country,
                                         value='America',
                                         inplace=True)
```

Now that we have all the data we want in a single `DataFrame`, we will reshape it back into panel form with a `MultiIndex`.

We should also ensure to sort the index using `.sort_index()` so that we can efficiently filter our dataframe later on.

By default, levels will be sorted top-down

```
In [22]: merged = merged.set_index(['Continent', 'Country']).sort_index()
         merged.head()
```

```
Out[22]:                      2006-01-01  2007-01-01  2008-01-01  …  2014-01-01  \
         Continent Country                                          …
         America   Brazil            0.87        0.92        0.96  …        1.21
                   Canada            6.89        6.96        7.24  …        8.22
                   Chile             1.42        1.45        1.44  …        1.76
                   Colombia          1.01        1.02        1.01  …        1.13
                   Costa Rica         nan         nan         nan  …        2.41
```

```
                        2015-01-01  2016-01-01
Continent Country
America   Brazil            1.21        1.24
          Canada            8.35        8.48
          Chile             1.81        1.91
          Colombia          1.13        1.12
          Costa Rica        2.56        2.63

[5 rows x 11 columns]
```

While merging, we lost our `DatetimeIndex`, as we merged columns that were not in datetime format

```
In [23]: merged.columns
```

```
Out[23]: Index([2006-01-01 00:00:00, 2007-01-01 00:00:00, 2008-01-01 00:00:00,
                2009-01-01 00:00:00, 2010-01-01 00:00:00, 2011-01-01 00:00:00,
                2012-01-01 00:00:00, 2013-01-01 00:00:00, 2014-01-01 00:00:00,
                2015-01-01 00:00:00, 2016-01-01 00:00:00],
               dtype='object')
```

Now that we have set the merged columns as the index, we can recreate a `DatetimeIndex` using `.to_datetime()`

```
In [24]: merged.columns = pd.to_datetime(merged.columns)
         merged.columns = merged.columns.rename('Time')
         merged.columns
```

```
Out[24]: DatetimeIndex(['2006-01-01', '2007-01-01', '2008-01-01', '2009-01-01',
                        '2010-01-01', '2011-01-01', '2012-01-01', '2013-01-01',
                        '2014-01-01', '2015-01-01', '2016-01-01'],
                       dtype='datetime64[ns]', name='Time', freq=None)
```

The `DatetimeIndex` tends to work more smoothly in the row axis, so we will go ahead and transpose `merged`

```
In [25]: merged = merged.transpose()
         merged.head()
```

```
Out[25]: Continent  America                 …   Europe
         Country     Brazil Canada Chile    … Slovenia Spain United Kingdom
         Time                               …
         2006-01-01    0.87   6.89  1.42    …     3.92  3.99           9.81
         2007-01-01    0.92   6.96  1.45    …     3.88  4.10          10.07
         2008-01-01    0.96   7.24  1.44    …     3.96  4.14          10.04
         2009-01-01    1.03   7.67  1.52    …     4.08  4.32          10.15
         2010-01-01    1.08   7.94  1.56    …     4.81  4.30           9.96

         [5 rows x 32 columns]
```

65.5 Grouping and Summarizing Data

Grouping and summarizing data can be particularly useful for understanding large panel datasets.

A simple way to summarize data is to call an aggregation method on the dataframe, such as `.mean()` or `.max()`.

For example, we can calculate the average real minimum wage for each country over the period 2006 to 2016 (the default is to aggregate over rows)

```
In [26]: merged.mean().head(10)
```

```
Out[26]: Continent  Country
         America    Brazil          1.09
                    Canada          7.82
                    Chile           1.62
                    Colombia        1.07
                    Costa Rica      2.53
                    Mexico          0.53
                    United States   7.15
         Asia       Israel          5.95
                    Japan           6.18
                    Korea           4.22
         dtype: float64
```

Using this series, we can plot the average real minimum wage over the past decade for each country in our data set

```
In [27]: import matplotlib.pyplot as plt
         %matplotlib inline
         import matplotlib
         matplotlib.style.use('seaborn')

         merged.mean().sort_values(ascending=False).plot(kind='bar',
 ↪title="Average real minimum
         wage 2006 - 2016")

         #Set country labels
         country_labels =
         merged.mean().sort_values(ascending=False).index.
 ↪get_level_values('Country').tolist()
         plt.xticks(range(0, len(country_labels)), country_labels)
         plt.xlabel('Country')

         plt.show()
```

Passing in `axis=1` to `.mean()` will aggregate over columns (giving the average minimum wage for all countries over time)

```
In [28]: merged.mean(axis=1).head()
```

```
Out[28]: Time
         2006-01-01   4.69
         2007-01-01   4.84
         2008-01-01   4.90
         2009-01-01   5.08
         2010-01-01   5.11
         dtype: float64
```

We can plot this time series as a line graph

```
In [29]: merged.mean(axis=1).plot()
         plt.title('Average real minimum wage 2006 - 2016')
         plt.ylabel('2015 USD')
         plt.xlabel('Year')
         plt.show()
```

We can also specify a level of the `MultiIndex` (in the column axis) to aggregate over

```
In [30]: merged.mean(level='Continent', axis=1).head()
```

```
Out[30]: Continent   America  Asia  Australia  Europe
         Time
         2006-01-01     2.80  4.29      10.25    4.80
         2007-01-01     2.85  4.44      10.73    4.94
         2008-01-01     2.99  4.45      10.76    4.99
         2009-01-01     3.23  4.53      10.97    5.16
         2010-01-01     3.34  4.53      10.95    5.17
```

We can plot the average minimum wages in each continent as a time series

```
In [31]: merged.mean(level='Continent', axis=1).plot()
         plt.title('Average real minimum wage')
         plt.ylabel('2015 USD')
         plt.xlabel('Year')
         plt.show()
```

We will drop Australia as a continent for plotting purposes

```
In [32]: merged = merged.drop('Australia', level='Continent', axis=1)
         merged.mean(level='Continent', axis=1).plot()
         plt.title('Average real minimum wage')
         plt.ylabel('2015 USD')
         plt.xlabel('Year')
         plt.show()
```

`.describe()` is useful for quickly retrieving a number of common summary statistics

```
In [33]: merged.stack().describe()
```

```
Out[33]: Continent  America  Asia  Europe
         count        69.00 44.00  200.00
         mean          3.19  4.70    5.15
         std           3.02  1.56    3.82
         min           0.52  2.22    0.23
         25%           1.03  3.37    2.02
         50%           1.44  5.48    3.54
         75%           6.96  5.95    9.70
         max           8.48  6.65   12.39
```

This is a simplified way to use `groupby`.

Using `groupby` generally follows a 'split-apply-combine' process:

- split: data is grouped based on one or more keys
- apply: a function is called on each group independently
- combine: the results of the function calls are combined into a new data structure

The `groupby` method achieves the first step of this process, creating a new `DataFrameGroupBy` object with data split into groups.

Let's split `merged` by continent again, this time using the `groupby` function, and name the resulting object `grouped`

```
In [34]: grouped = merged.groupby(level='Continent', axis=1)
         grouped
```

```
Out[34]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f869bf5a8d0>
```

Calling an aggregation method on the object applies the function to each group, the results of which are combined in a new data structure.

For example, we can return the number of countries in our dataset for each continent using `.size()`.

In this case, our new data structure is a `Series`

```
In [35]: grouped.size()
```

```
Out[35]: Continent
         America     7
         Asia        4
         Europe     19
         dtype: int64
```

Calling `.get_group()` to return just the countries in a single group, we can create a kernel density estimate of the distribution of real minimum wages in 2016 for each continent.

`grouped.groups.keys()` will return the keys from the `groupby` object

```
In [36]: import seaborn as sns

         continents = grouped.groups.keys()

         for continent in continents:
             sns.kdeplot(grouped.get_group(continent)['2015'].unstack(),
↪label=continent,
         shade=True)

         plt.title('Real minimum wages in 2015')
         plt.xlabel('US dollars')
         plt.show()
```

65.6 Final Remarks

This lecture has provided an introduction to some of pandas' more advanced features, including multiindices, merging, grouping and plotting.

Other tools that may be useful in panel data analysis include xarray, a python package that extends pandas to N-dimensional data structures.

65.7 Exercises

65.7.1 Exercise 1

In these exercises, you'll work with a dataset of employment rates in Europe by age and sex from Eurostat.

The dataset can be accessed with the following link:

```
In [37]: url3 = 'https://raw.githubusercontent.com/QuantEcon/lecture-source-
         py/master/source/_static/lecture_specific/pandas_panel/employ.csv'
```

Reading in the CSV file returns a panel dataset in long format. Use `.pivot_table()` to construct a wide format dataframe with a `MultiIndex` in the columns.

Start off by exploring the dataframe and the variables available in the `MultiIndex` levels.

Write a program that quickly returns all values in the `MultiIndex`.

65.7.2 Exercise 2

Filter the above dataframe to only include employment as a percentage of 'active population'.

Create a grouped boxplot using `seaborn` of employment rates in 2015 by age group and sex.

Hint: `GEO` includes both areas and countries.

65.8 Solutions

65.8.1 Exercise 1

```
In [38]: employ = pd.read_csv(url3)
         employ = employ.pivot_table(values='Value',
                                     index=['DATE'],
                                 columns=['UNIT','AGE', 'SEX', 'INDIC_EM', 'GEO'])
         employ.index = pd.to_datetime(employ.index) # ensure that dates are
 ↪datetime format
         employ.head()
```

```
Out[38]: UNIT       Percentage of total population                    …  \
         AGE                    From 15 to 24 years                    …
         SEX                                Females                    …
         INDIC_EM                 Active population                    …
         GEO                              Austria Belgium Bulgaria     …
         DATE                                                          …
         2007-01-01                         56.00   31.60    26.00    …
         2008-01-01                         56.20   30.80    26.10    …
         2009-01-01                         56.20   29.90    24.80    …
         2010-01-01                         54.00   29.80    26.60    …
         2011-01-01                         54.80   29.80    24.80    …

         UNIT                                          Thousand persons          \
         AGE                                         From 55 to 64 years
         SEX                                                       Total
         INDIC_EM   Total employment (resident population concept - LFS)
         GEO                                                Switzerland   Turkey
         DATE
         2007-01-01                                                 nan   1,282.00
         2008-01-01                                                 nan   1,354.00
         2009-01-01                                                 nan   1,449.00
         2010-01-01                                              640.00   1,583.00
```

```
2011-01-01                                                661.00   1,760.00

UNIT
AGE
SEX
INDIC_EM
GEO        United Kingdom
DATE
2007-01-01       4,131.00
2008-01-01       4,204.00
2009-01-01       4,193.00
2010-01-01       4,186.00
2011-01-01       4,164.00

[5 rows x 1440 columns]
```

This is a large dataset so it is useful to explore the levels and variables available

```
In [39]: employ.columns.names
```

```
Out[39]: FrozenList(['UNIT', 'AGE', 'SEX', 'INDIC_EM', 'GEO'])
```

Variables within levels can be quickly retrieved with a loop

```
In [40]: for name in employ.columns.names:
             print(name, employ.columns.get_level_values(name).unique())
```

```
    UNIT Index(['Percentage of total population', 'Thousand persons'],
 ↪dtype='object',
name='UNIT')
AGE Index(['From 15 to 24 years', 'From 25 to 54 years', 'From 55 to 64 years'],
dtype='object', name='AGE')
SEX Index(['Females', 'Males', 'Total'], dtype='object', name='SEX')
INDIC_EM Index(['Active population', 'Total employment (resident population concept
 ↪-
LFS)'], dtype='object', name='INDIC_EM')
GEO Index(['Austria', 'Belgium', 'Bulgaria', 'Croatia', 'Cyprus', 'Czech Republic',
       'Denmark', 'Estonia', 'Euro area (17 countries)',
       'Euro area (18 countries)', 'Euro area (19 countries)',
       'European Union (15 countries)', 'European Union (27 countries)',
       'European Union (28 countries)', 'Finland',
       'Former Yugoslav Republic of Macedonia, the', 'France',
       'France (metropolitan)',
       'Germany (until 1990 former territory of the FRG)', 'Greece', 'Hungary',
       'Iceland', 'Ireland', 'Italy', 'Latvia', 'Lithuania', 'Luxembourg',
       'Malta', 'Netherlands', 'Norway', 'Poland', 'Portugal', 'Romania',
       'Slovakia', 'Slovenia', 'Spain', 'Sweden', 'Switzerland', 'Turkey',
       'United Kingdom'],
      dtype='object', name='GEO')
```

65.8.2 Exercise 2

To easily filter by country, swap `GEO` to the top level and sort the `MultiIndex`

```
In [41]: employ.columns = employ.columns.swaplevel(0,-1)
         employ = employ.sort_index(axis=1)
```

We need to get rid of a few items in `GEO` which are not countries.

A fast way to get rid of the EU areas is to use a list comprehension to find the level values in `GEO` that begin with 'Euro'

```
In [42]: geo_list = employ.columns.get_level_values('GEO').unique().tolist()
         countries = [x for x in geo_list if not x.startswith('Euro')]
         employ = employ[countries]
         employ.columns.get_level_values('GEO').unique()
```

```
Out[42]: Index(['Austria', 'Belgium', 'Bulgaria', 'Croatia', 'Cyprus', 'Czech
 ↪Republic',
                'Denmark', 'Estonia', 'Finland',
                'Former Yugoslav Republic of Macedonia, the', 'France',
                'France (metropolitan)',
                'Germany (until 1990 former territory of the FRG)', 'Greece',
 ↪'Hungary',
                'Iceland', 'Ireland', 'Italy', 'Latvia', 'Lithuania', 'Luxembourg',
                'Malta', 'Netherlands', 'Norway', 'Poland', 'Portugal', 'Romania',
                'Slovakia', 'Slovenia', 'Spain', 'Sweden', 'Switzerland', 'Turkey',
                'United Kingdom'],
               dtype='object', name='GEO')
```

Select only percentage employed in the active population from the dataframe

```
In [43]: employ_f = employ.xs(('Percentage of total population', 'Active
 ↪population'),
                               level=('UNIT', 'INDIC_EM'),
                               axis=1)
         employ_f.head()
```

```
Out[43]: GEO                    Austria                …      United Kingdom        \
         AGE        From 15 to 24 years                … From 55 to 64 years
         SEX                    Females Males Total    …              Females Males
         DATE                                          …
         2007-01-01               56.00 62.90 59.40   …                49.90 68.90
         2008-01-01               56.20 62.90 59.50   …                50.20 69.80
         2009-01-01               56.20 62.90 59.50   …                50.60 70.30
         2010-01-01               54.00 62.60 58.30   …                51.10 69.20
         2011-01-01               54.80 63.60 59.20   …                51.30 68.40

         GEO
         AGE
         SEX        Total
         DATE
         2007-01-01 59.30
         2008-01-01 59.80
         2009-01-01 60.30
         2010-01-01 60.00
         2011-01-01 59.70

         [5 rows x 306 columns]
```

Drop the 'Total' value before creating the grouped boxplot

```
In [44]: employ_f = employ_f.drop('Total', level='SEX', axis=1)
```

```
In [45]: box = employ_f['2015'].unstack().reset_index()
         sns.boxplot(x="AGE", y=0, hue="SEX", data=box, palette=("husl"),
 ↪showfliers=False)
         plt.xlabel('')
         plt.xticks(rotation=35)
         plt.ylabel('Percentage of population (%)')
         plt.title('Employment in Europe (2015)')
         plt.legend(bbox_to_anchor=(1,0.5))
         plt.show()
```

Chapter 66

Linear Regression in Python

66.1 Contents

- Overview 66.2
- Simple Linear Regression 66.3
- Extending the Linear Regression Model 66.4
- Endogeneity 66.5
- Summary 66.6
- Exercises 66.7
- Solutions 66.8

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install linearmodels
```

66.2 Overview

Linear regression is a standard tool for analyzing the relationship between two or more variables.

In this lecture, we'll use the Python package `statsmodels` to estimate, interpret, and visualize linear regression models.

Along the way, we'll discuss a variety of topics, including

- simple and multivariate linear regression
- visualization
- endogeneity and omitted variable bias
- two-stage least squares

As an example, we will replicate results from Acemoglu, Johnson and Robinson's seminal paper [3].

- You can download a copy here.

In the paper, the authors emphasize the importance of institutions in economic development.

The main contribution is the use of settler mortality rates as a source of *exogenous* variation in institutional differences.

Such variation is needed to determine whether it is institutions that give rise to greater economic growth, rather than the other way around.

Let's start with some imports:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        import pandas as pd
        import statsmodels.api as sm
        from statsmodels.iolib.summary2 import summary_col
        from linearmodels.iv import IV2SLS
```

66.2.1 Prerequisites

This lecture assumes you are familiar with basic econometrics.

For an introductory text covering these topics, see, for example, [165].

66.2.2 Comments

This lecture is coauthored with Natasha Watkins.

66.3 Simple Linear Regression

[3] wish to determine whether or not differences in institutions can help to explain observed economic outcomes.

How do we measure *institutional differences* and *economic outcomes*?

In this paper,

- economic outcomes are proxied by log GDP per capita in 1995, adjusted for exchange rates.
- institutional differences are proxied by an index of protection against expropriation on average over 1985-95, constructed by the Political Risk Services Group.

These variables and other data used in the paper are available for download on Daron Acemoglu's webpage.

We will use pandas' `.read_stata()` function to read in data contained in the `.dta` files to dataframes

```
In [3]: df1 = pd.read_stata('https://github.com/QuantEcon/lecture-source-
        py/blob/master/source/_static/lecture_specific/ols/maketable1.
 ↪dta?raw=true')
        df1.head()
```

```
Out[3]:   shortnam   euro1900  excolony    avexpr  logpgp95  cons1  cons90 
 ↪democ00a  \
       0      AFG   0.000000       1.0       NaN       NaN    1.0     2.0       1.0
       1      AGO   8.000000       1.0  5.363636  7.770645    3.0     3.0       0.0
       2      ARE   0.000000       1.0  7.181818  9.804219    NaN     NaN       NaN
```

```
3      ARG  60.000004       1.0  6.386364  9.133459    1.0     6.0       3.0
4      ARM   0.000000       0.0       NaN  7.682482    NaN     NaN       NaN

   cons00a    extmort4    logem4  loghjypl  baseco
0      1.0   93.699997  4.540098       NaN     NaN
1      1.0  280.000000  5.634789 -3.411248     1.0
2      NaN         NaN       NaN       NaN     NaN
3      3.0   68.900002  4.232656 -0.872274     1.0
4      NaN         NaN       NaN       NaN     NaN
```

Let's use a scatterplot to see whether any obvious relationship exists between GDP per capita and the protection against expropriation index

```
In [4]: plt.style.use('seaborn')

        df1.plot(x='avexpr', y='logpgp95', kind='scatter')
        plt.show()
```

The plot shows a fairly strong positive relationship between protection against expropriation and log GDP per capita.

Specifically, if higher protection against expropriation is a measure of institutional quality, then better institutions appear to be positively correlated with better economic outcomes (higher GDP per capita).

Given the plot, choosing a linear model to describe this relationship seems like a reasonable assumption.

We can write our model as

$$logpgp95_i = \beta_0 + \beta_1 avexpr_i + u_i$$

where:

- β_0 is the intercept of the linear trend line on the y-axis
- β_1 is the slope of the linear trend line, representing the *marginal effect* of protection against risk on log GDP per capita
- u_i is a random error term (deviations of observations from the linear trend due to factors not included in the model)

Visually, this linear model involves choosing a straight line that best fits the data, as in the following plot (Figure 2 in [3])

In [5]:
```
# Dropping NA's is required to use numpy's polyfit
df1_subset = df1.dropna(subset=['logpgp95', 'avexpr'])

# Use only 'base sample' for plotting purposes
df1_subset = df1_subset[df1_subset['baseco'] == 1]

X = df1_subset['avexpr']
y = df1_subset['logpgp95']
labels = df1_subset['shortnam']

# Replace markers with country labels
fig, ax = plt.subplots()
ax.scatter(X, y, marker='')

for i, label in enumerate(labels):
    ax.annotate(label, (X.iloc[i], y.iloc[i]))

# Fit a linear trend line
ax.plot(np.unique(X),
         np.poly1d(np.polyfit(X, y, 1))(np.unique(X)),
         color='black')

ax.set_xlim([3.3,10.5])
ax.set_ylim([4,10.5])
ax.set_xlabel('Average Expropriation Risk 1985-95')
ax.set_ylabel('Log GDP per capita, PPP, 1995')
ax.set_title('Figure 2: OLS relationship between expropriation \
    risk and income')
plt.show()
```

The most common technique to estimate the parameters (β's) of the linear model is Ordinary Least Squares (OLS).

As the name implies, an OLS model is solved by finding the parameters that minimize *the sum of squared residuals*, i.e.

$$\min_{\hat{\beta}} \sum_{i=1}^{N} \hat{u}_i^2$$

where $\hat{u}_i$ is the difference between the observation and the predicted value of the dependent variable.

To estimate the constant term β_0, we need to add a column of 1's to our dataset (consider the equation if β_0 was replaced with $\beta_0 x_i$ and $x_i = 1$)

```
In [6]: df1['const'] = 1
```

Now we can construct our model in `statsmodels` using the OLS function.

We will use `pandas` dataframes with `statsmodels`, however standard arrays can also be used as arguments

```
In [7]: reg1 = sm.OLS(endog=df1['logpgp95'], exog=df1[['const', 'avexpr']], \
            missing='drop')
        type(reg1)
```

```
Out[7]: statsmodels.regression.linear_model.OLS
```

So far we have simply constructed our model.

We need to use `.fit()` to obtain parameter estimates $\hat{\beta}_0$ and $\hat{\beta}_1$

```
In [8]: results = reg1.fit()
        type(results)
```

```
Out[8]: statsmodels.regression.linear_model.RegressionResultsWrapper
```

We now have the fitted regression model stored in `results`.

To view the OLS regression results, we can call the `.summary()` method.

Note that an observation was mistakenly dropped from the results in the original paper (see the note located in maketable2.do from Acemoglu's webpage), and thus the coefficients differ slightly.

```
In [9]: print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:               logpgp95   R-squared:                       0.611
Model:                            OLS   Adj. R-squared:                  0.608
Method:                 Least Squares   F-statistic:                     171.4
Date:                Tue, 03 Mar 2020   Prob (F-statistic):           4.16e-24
Time:                        17:10:18   Log-Likelihood:                -119.71
No. Observations:                 111   AIC:                             243.4
Df Residuals:                     109   BIC:                             248.8
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          4.6261      0.301     15.391      0.000       4.030       5.222
avexpr         0.5319      0.041     13.093      0.000       0.451       0.612
==============================================================================
Omnibus:                        9.251   Durbin-Watson:                   1.689
Prob(Omnibus):                  0.010   Jarque-Bera (JB):                9.170
Skew:                          -0.680   Prob(JB):                       0.0102
Kurtosis:                       3.362   Cond. No.                         33.2
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

From our results, we see that

- The intercept $\hat{\beta}_0 = 4.63$.
- The slope $\hat{\beta}_1 = 0.53$.
- The positive $\hat{\beta}_1$ parameter estimate implies that. institutional quality has a positive effect on economic outcomes, as we saw in the figure.
- The p-value of 0.000 for $\hat{\beta}_1$ implies that the effect of institutions on GDP is statistically significant (using $p < 0.05$ as a rejection rule).
- The R-squared value of 0.611 indicates that around 61% of variation in log GDP per capita is explained by protection against expropriation.

Using our parameter estimates, we can now write our estimated relationship as

$$\widehat{logpgp95}_i = 4.63 + 0.53\ avexpr_i$$

This equation describes the line that best fits our data, as shown in Figure 2.

We can use this equation to predict the level of log GDP per capita for a value of the index of expropriation protection.

For example, for a country with an index value of 7.07 (the average for the dataset), we find that their predicted level of log GDP per capita in 1995 is 8.38.

```
In [10]: mean_expr = np.mean(df1_subset['avexpr'])
         mean_expr
```

```
Out[10]: 6.515625
```

```
In [11]: predicted_logpdp95 = 4.63 + 0.53 * 7.07
         predicted_logpdp95
```

```
Out[11]: 8.3771
```

An easier (and more accurate) way to obtain this result is to use `.predict()` and set $constant = 1$ and $avexpr_i = mean_expr$

```
In [12]: results.predict(exog=[1, mean_expr])
```

```
Out[12]: array([8.09156367])
```

We can obtain an array of predicted $logpgp95_i$ for every value of $avexpr_i$ in our dataset by calling `.predict()` on our results.

Plotting the predicted values against $avexpr_i$ shows that the predicted values lie along the linear line that we fitted above.

The observed values of $logpgp95_i$ are also plotted for comparison purposes

```
In [13]: # Drop missing observations from whole sample

         df1_plot = df1.dropna(subset=['logpgp95', 'avexpr'])

         # Plot predicted values

         fix, ax = plt.subplots()
         ax.scatter(df1_plot['avexpr'], results.predict(), alpha=0.5,
                    label='predicted')

         # Plot observed values

         ax.scatter(df1_plot['avexpr'], df1_plot['logpgp95'], alpha=0.5,
                    label='observed')

         ax.legend()
```

```
ax.set_title('OLS predicted values')
ax.set_xlabel('avexpr')
ax.set_ylabel('logpgp95')
plt.show()
```

66.4 Extending the Linear Regression Model

So far we have only accounted for institutions affecting economic performance - almost certainly there are numerous other factors affecting GDP that are not included in our model.

Leaving out variables that affect $logpgp95_i$ will result in **omitted variable bias**, yielding biased and inconsistent parameter estimates.

We can extend our bivariate regression model to a **multivariate regression model** by adding in other factors that may affect $logpgp95_i$.

[3] consider other factors such as:

- the effect of climate on economic outcomes; latitude is used to proxy this
- differences that affect both economic performance and institutions, eg. cultural, historical, etc.; controlled for with the use of continent dummies

Let's estimate some of the extended models considered in the paper (Table 2) using data from `maketable2.dta`

```
In [14]: df2 = pd.read_stata('https://github.com/QuantEcon/lecture-source-
         py/blob/master/source/_static/lecture_specific/ols/maketable2.
↪dta?raw=true')
```

```
# Add constant term to dataset
df2['const'] = 1

# Create lists of variables to be used in each regression
X1 = ['const', 'avexpr']
X2 = ['const', 'avexpr', 'lat_abst']
X3 = ['const', 'avexpr', 'lat_abst', 'asia', 'africa', 'other']

# Estimate an OLS regression for each set of variables
reg1 = sm.OLS(df2['logpgp95'], df2[X1], missing='drop').fit()
reg2 = sm.OLS(df2['logpgp95'], df2[X2], missing='drop').fit()
reg3 = sm.OLS(df2['logpgp95'], df2[X3], missing='drop').fit()
```

Now that we have fitted our model, we will use `summary_col` to display the results in a single table (model numbers correspond to those in the paper)

```
In [15]: info_dict={'R-squared' : lambda x: f"{x.rsquared:.2f}",
                    'No. observations' : lambda x: f"{int(x.nobs):d}"}

         results_table = summary_col(results=[reg1,reg2,reg3],
                                     float_format='%0.2f',
                                     stars = True,
                                     model_names=['Model 1',
                                                  'Model 3',
                                                  'Model 4'],
                                     info_dict=info_dict,
                                     regressor_order=['const',
                                                      'avexpr',
                                                      'lat_abst',
                                                      'asia',
                                                      'africa'])

         results_table.add_title('Table 2 - OLS Regressions')

         print(results_table)
```

```
            Table 2 - OLS Regressions
=========================================
                 Model 1 Model 3 Model 4
-----------------------------------------
const            4.63*** 4.87*** 5.85***
                 (0.30)  (0.33)  (0.34)
avexpr           0.53*** 0.46*** 0.39***
                 (0.04)  (0.06)  (0.05)
lat_abst                 0.87*   0.33
                         (0.49)  (0.45)
asia                             -0.15
                                 (0.15)
africa                           -0.92***
                                 (0.17)
other                            0.30
                                 (0.37)
R-squared        0.61    0.62    0.72
No. observations 111     111     111
=========================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
```

66.5 Endogeneity

As [3] discuss, the OLS models likely suffer from **endogeneity** issues, resulting in biased and inconsistent model estimates.

Namely, there is likely a two-way relationship between institutions and economic outcomes:

- richer countries may be able to afford or prefer better institutions
- variables that affect income may also be correlated with institutional differences
- the construction of the index may be biased; analysts may be biased towards seeing countries with higher income having better institutions

To deal with endogeneity, we can use **two-stage least squares (2SLS) regression**, which is an extension of OLS regression.

This method requires replacing the endogenous variable $avexpr_i$ with a variable that is:

1. correlated with $avexpr_i$
2. not correlated with the error term (ie. it should not directly affect the dependent variable, otherwise it would be correlated with u_i due to omitted variable bias)

The new set of regressors is called an **instrument**, which aims to remove endogeneity in our proxy of institutional differences.

The main contribution of [3] is the use of settler mortality rates to instrument for institutional differences.

They hypothesize that higher mortality rates of colonizers led to the establishment of institutions that were more extractive in nature (less protection against expropriation), and these institutions still persist today.

Using a scatterplot (Figure 3 in [3]), we can see protection against expropriation is negatively correlated with settler mortality rates, coinciding with the authors' hypothesis and satisfying the first condition of a valid instrument.

In [16]:
```
# Dropping NA's is required to use numpy's polyfit
df1_subset2 = df1.dropna(subset=['logem4', 'avexpr'])

X = df1_subset2['logem4']
y = df1_subset2['avexpr']
labels = df1_subset2['shortnam']

# Replace markers with country labels
fig, ax = plt.subplots()
ax.scatter(X, y, marker='')

for i, label in enumerate(labels):
    ax.annotate(label, (X.iloc[i], y.iloc[i]))

# Fit a linear trend line
ax.plot(np.unique(X),
         np.poly1d(np.polyfit(X, y, 1))(np.unique(X)),
         color='black')

ax.set_xlim([1.8,8.4])
ax.set_ylim([3.3,10.4])
```

```
    ax.set_xlabel('Log of Settler Mortality')
    ax.set_ylabel('Average Expropriation Risk 1985-95')
    ax.set_title('Figure 3: First-stage relationship between settler
↪mortality \
            and expropriation risk')
    plt.show()
```

The second condition may not be satisfied if settler mortality rates in the 17th to 19th centuries have a direct effect on current GDP (in addition to their indirect effect through institutions).

For example, settler mortality rates may be related to the current disease environment in a country, which could affect current economic performance.

[3] argue this is unlikely because:

- The majority of settler deaths were due to malaria and yellow fever and had a limited effect on local people.
- The disease burden on local people in Africa or India, for example, did not appear to be higher than average, supported by relatively high population densities in these areas before colonization.

As we appear to have a valid instrument, we can use 2SLS regression to obtain consistent and unbiased parameter estimates.

First stage

The first stage involves regressing the endogenous variable ($avexpr_i$) on the instrument.

The instrument is the set of all exogenous variables in our model (and not just the variable we have replaced).

Using model 1 as an example, our instrument is simply a constant and settler mortality rates

$logem4_i$.

Therefore, we will estimate the first-stage regression as

$$avexpr_i = \delta_0 + \delta_1 logem4_i + v_i$$

The data we need to estimate this equation is located in `maketable4.dta` (only complete data, indicated by `baseco = 1`, is used for estimation)

```
In [17]: # Import and select the data
         df4 = pd.read_stata('https://github.com/QuantEcon/lecture-source-
         py/blob/master/source/_static/lecture_specific/ols/maketable4.
↪dta?raw=true')
         df4 = df4[df4['baseco'] == 1]

         # Add a constant variable
         df4['const'] = 1

         # Fit the first stage regression and print summary
         results_fs = sm.OLS(df4['avexpr'],
                             df4[['const', 'logem4']],
                             missing='drop').fit()
         print(results_fs.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 avexpr   R-squared:                       0.270
Model:                            OLS   Adj. R-squared:                  0.258
Method:                 Least Squares   F-statistic:                     22.95
Date:                Tue, 03 Mar 2020   Prob (F-statistic):           1.08e-05
Time:                        17:10:21   Log-Likelihood:                -104.83
No. Observations:                  64   AIC:                             213.7
Df Residuals:                      62   BIC:                             218.0
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          9.3414      0.611     15.296      0.000       8.121      10.562
logem4        -0.6068      0.127     -4.790      0.000      -0.860      -0.354
==============================================================================
Omnibus:                        0.035   Durbin-Watson:                   2.003
Prob(Omnibus):                  0.983   Jarque-Bera (JB):                0.172
Skew:                           0.045   Prob(JB):                        0.918
Kurtosis:                       2.763   Cond. No.                         19.4
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

Second stage

We need to retrieve the predicted values of $avexpr_i$ using `.predict()`.

We then replace the endogenous variable $avexpr_i$ with the predicted values $\widehat{avexpr}_i$ in the original linear model.

Our second stage regression is thus

$$logpgp95_i = \beta_0 + \beta_1 \widehat{avexpr}_i + u_i$$

```
In [18]: df4['predicted_avexpr'] = results_fs.predict()

         results_ss = sm.OLS(df4['logpgp95'],
                             df4[['const', 'predicted_avexpr']]).fit()
         print(results_ss.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:               logpgp95   R-squared:                       0.477
Model:                            OLS   Adj. R-squared:                  0.469
Method:                 Least Squares   F-statistic:                     56.60
Date:                Tue, 03 Mar 2020   Prob (F-statistic):           2.66e-10
Time:                        17:10:21   Log-Likelihood:                -72.268
No. Observations:                  64   AIC:                             148.5
Df Residuals:                      62   BIC:                             152.9
Df Model:                           1
Covariance Type:            nonrobust
=================================================================================
                      coef    std err          t      P>|t|      [0.025      0.975]
---------------------------------------------------------------------------------
-
const                1.9097      0.823      2.320      0.024       0.264       3.555
predicted_avexpr     0.9443      0.126      7.523      0.000       0.693       1.195
==============================================================================
Omnibus:                       10.547   Durbin-Watson:                   2.137
Prob(Omnibus):                  0.005   Jarque-Bera (JB):               11.010
Skew:                          -0.790   Prob(JB):                      0.00407
Kurtosis:                       4.277   Cond. No.                         58.1
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

The second-stage regression results give us an unbiased and consistent estimate of the effect of institutions on economic outcomes.

The result suggests a stronger positive relationship than what the OLS results indicated.

Note that while our parameter estimates are correct, our standard errors are not and for this reason, computing 2SLS 'manually' (in stages with OLS) is not recommended.

We can correctly estimate a 2SLS regression in one step using the linearmodels package, an extension of `statsmodels`

Note that when using `IV2SLS`, the exogenous and instrument variables are split up in the function arguments (whereas before the instrument included exogenous variables)

```
In [19]: iv = IV2SLS(dependent=df4['logpgp95'],
                     exog=df4['const'],
                     endog=df4['avexpr'],
                     instruments=df4['logem4']).fit(cov_type='unadjusted')

         print(iv.summary)
```

```
                            IV-2SLS Estimation Summary
==============================================================================
Dep. Variable:               logpgp95   R-squared:                      0.1870
Estimator:                    IV-2SLS   Adj. R-squared:                 0.1739
No. Observations:                  64   F-statistic:                    37.568
Date:                Tue, Mar 03 2020   P-value (F-stat)                0.0000
Time:                        17:10:21   Distribution:                  chi2(1)
Cov. Estimator:            unadjusted

                            Parameter Estimates
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
const          1.9097     1.0106     1.8897     0.0588     -0.0710      3.8903
avexpr         0.9443     0.1541     6.1293     0.0000      0.6423      1.2462
==============================================================================

Endogenous: avexpr
Instruments: logem4
Unadjusted Covariance (Homoskedastic)
Debiased: False
```

Given that we now have consistent and unbiased estimates, we can infer from the model we have estimated that institutional differences (stemming from institutions set up during colonization) can help to explain differences in income levels across countries today.

[3] use a marginal effect of 0.94 to calculate that the difference in the index between Chile and Nigeria (ie. institutional quality) implies up to a 7-fold difference in income, emphasizing the significance of institutions in economic development.

66.6 Summary

We have demonstrated basic OLS and 2SLS regression in `statsmodels` and `linearmodels`.

If you are familiar with R, you may want to use the formula interface to `statsmodels`, or consider using r2py to call R from within Python.

66.7 Exercises

66.7.1 Exercise 1

In the lecture, we think the original model suffers from endogeneity bias due to the likely effect income has on institutional development.

Although endogeneity is often best identified by thinking about the data and model, we can formally test for endogeneity using the **Hausman test**.

We want to test for correlation between the endogenous variable, $avexpr_i$, and the errors, u_i

$$H_0 : Cov(avexpr_i, u_i) = 0 \quad (no\ endogeneity)$$
$$H_1 : Cov(avexpr_i, u_i) \neq 0 \quad (endogeneity)$$

This test is running in two stages.

First, we regress $avexpr_i$ on the instrument, $logem4_i$

$$avexpr_i = \pi_0 + \pi_1 logem4_i + \upsilon_i$$

Second, we retrieve the residuals $\hat{\upsilon}_i$ and include them in the original equation

$$logpgp95_i = \beta_0 + \beta_1 avexpr_i + \alpha\hat{\upsilon}_i + u_i$$

If α is statistically significant (with a p-value < 0.05), then we reject the null hypothesis and conclude that $avexpr_i$ is endogenous.

Using the above information, estimate a Hausman test and interpret your results.

66.7.2 Exercise 2

The OLS parameter β can also be estimated using matrix algebra and `numpy` (you may need to review the numpy lecture to complete this exercise).

The linear equation we want to estimate is (written in matrix form)

$$y = X\beta + u$$

To solve for the unknown parameter β, we want to minimize the sum of squared residuals

$$\min_{\hat{\beta}} \hat{u}'\hat{u}$$

Rearranging the first equation and substituting into the second equation, we can write

$$\min_{\hat{\beta}} (Y - X\hat{\beta})'(Y - X\hat{\beta})$$

Solving this optimization problem gives the solution for the $\hat{\beta}$ coefficients

$$\hat{\beta} = (X'X)^{-1}X'y$$

Using the above information, compute $\hat{\beta}$ from model 1 using `numpy` - your results should be the same as those in the `statsmodels` output from earlier in the lecture.

66.8 Solutions

66.8.1 Exercise 1

```
In [20]: # Load in data
         df4 = pd.read_stata('https://github.com/QuantEcon/lecture-source-
         py/blob/master/source/_static/lecture_specific/ols/maketable4.
↪dta?raw=true')

         # Add a constant term
```

```
df4['const'] = 1

# Estimate the first stage regression
reg1 = sm.OLS(endog=df4['avexpr'],
              exog=df4[['const', 'logem4']],
              missing='drop').fit()

# Retrieve the residuals
df4['resid'] = reg1.resid

# Estimate the second stage residuals
reg2 = sm.OLS(endog=df4['logpgp95'],
              exog=df4[['const', 'avexpr', 'resid']],
              missing='drop').fit()

print(reg2.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:               logpgp95   R-squared:                       0.689
Model:                            OLS   Adj. R-squared:                  0.679
Method:                 Least Squares   F-statistic:                     74.05
Date:                Tue, 03 Mar 2020   Prob (F-statistic):           1.07e-17
Time:                        17:10:21   Log-Likelihood:                -62.031
No. Observations:                  70   AIC:                             130.1
Df Residuals:                      67   BIC:                             136.8
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          2.4782      0.547      4.530      0.000       1.386       3.570
avexpr         0.8564      0.082     10.406      0.000       0.692       1.021
resid         -0.4951      0.099     -5.017      0.000      -0.692      -0.298
==============================================================================
Omnibus:                       17.597   Durbin-Watson:                   2.086
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               23.194
Skew:                          -1.054   Prob(JB):                     9.19e-06
Kurtosis:                       4.873   Cond. No.                         53.8
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

The output shows that the coefficient on the residuals is statistically significant, indicating $avexpr_i$ is endogenous.

66.8.2 Exercise 2

```
In [21]: # Load in data
         df1 = pd.read_stata('https://github.com/QuantEcon/lecture-source-
         py/blob/master/source/_static/lecture_specific/ols/maketable1.
↪dta?raw=true')
         df1 = df1.dropna(subset=['logpgp95', 'avexpr'])

         # Add a constant term
```

```
df1['const'] = 1

# Define the X and y variables
y = np.asarray(df1['logpgp95'])
X = np.asarray(df1[['const', 'avexpr']])

# Compute β_hat
β_hat = np.linalg.solve(X.T @ X, X.T @ y)

# Print out the results from the 2 x 1 vector β_hat
print(f'β_0 = {β_hat[0]:.2}')
print(f'β_1 = {β_hat[1]:.2}')
```

```
  β_0 = 4.6
β_1 = 0.53
```

It is also possible to use `np.linalg.inv(X.T @ X) @ X.T @ y` to solve for β, however `.solve()` is preferred as it involves fewer computations.

Chapter 67

Maximum Likelihood Estimation

67.1 Contents

- Overview 67.2
- Set Up and Assumptions 67.3
- Conditional Distributions 67.4
- Maximum Likelihood Estimation 67.5
- MLE with Numerical Methods 67.6
- Maximum Likelihood Estimation with `statsmodels` 67.7
- Summary 67.8
- Exercises 67.9
- Solutions 67.10

67.2 Overview

In a previous lecture, we estimated the relationship between dependent and explanatory variables using linear regression.

But what if a linear relationship is not an appropriate assumption for our model?

One widely used alternative is maximum likelihood estimation, which involves specifying a class of distributions, indexed by unknown parameters, and then using the data to pin down these parameter values.

The benefit relative to linear regression is that it allows more flexibility in the probabilistic relationships between variables.

Here we illustrate maximum likelihood by replicating Daniel Treisman's (2016) paper, Russia's Billionaires, which connects the number of billionaires in a country to its economic characteristics.

The paper concludes that Russia has a higher number of billionaires than economic factors such as market size and tax rate predict.

We'll require the following imports:

```
In [1]: import numpy as np
        from numpy import exp
        import matplotlib.pyplot as plt
```

```
%matplotlib inline
from scipy.special import factorial
import pandas as pd
from mpl_toolkits.mplot3d import Axes3D
import statsmodels.api as sm
from statsmodels.api import Poisson
from scipy import stats
from scipy.stats import norm
from statsmodels.iolib.summary2 import summary_col
```

67.2.1 Prerequisites

We assume familiarity with basic probability and multivariate calculus.

67.2.2 Comments

This lecture is co-authored with Natasha Watkins.

67.3 Set Up and Assumptions

Let's consider the steps we need to go through in maximum likelihood estimation and how they pertain to this study.

67.3.1 Flow of Ideas

The first step with maximum likelihood estimation is to choose the probability distribution believed to be generating the data.

More precisely, we need to make an assumption as to which *parametric class* of distributions is generating the data.

- e.g., the class of all normal distributions, or the class of all gamma distributions.

Each such class is a family of distributions indexed by a finite number of parameters.

- e.g., the class of normal distributions is a family of distributions indexed by its mean $\mu \in (-\infty, \infty)$ and standard deviation $\sigma \in (0, \infty)$.

We'll let the data pick out a particular element of the class by pinning down the parameters.

The parameter estimates so produced will be called **maximum likelihood estimates**.

67.3.2 Counting Billionaires

Treisman [157] is interested in estimating the number of billionaires in different countries.

The number of billionaires is integer-valued.

Hence we consider distributions that take values only in the nonnegative integers.

(This is one reason least squares regression is not the best tool for the present problem, since the dependent variable in linear regression is not restricted to integer values)

One integer distribution is the Poisson distribution, the probability mass function (pmf) of which is

$$f(y) = \frac{\mu^y}{y!} e^{-\mu}, \qquad y = 0, 1, 2, \ldots, \infty$$

We can plot the Poisson distribution over y for different values of μ as follows

```
In [2]: poisson_pmf = lambda y, μ: μ**y / factorial(y) * exp(-μ)
        y_values = range(0, 25)

        fig, ax = plt.subplots(figsize=(12, 8))

        for μ in [1, 5, 10]:
            distribution = []
            for y_i in y_values:
                distribution.append(poisson_pmf(y_i, μ))
            ax.plot(y_values,
                    distribution,
                    label=f'$\mu$={μ}',
                    alpha=0.5,
                    marker='o',
                    markersize=8)

        ax.grid()
        ax.set_xlabel('$y$', fontsize=14)
        ax.set_ylabel('$f(y \mid \mu)$', fontsize=14)
        ax.axis(xmin=0, ymin=0)
        ax.legend(fontsize=14)

        plt.show()
```

Notice that the Poisson distribution begins to resemble a normal distribution as the mean of y increases.

Let's have a look at the distribution of the data we'll be working with in this lecture.

Treisman's main source of data is *Forbes'* annual rankings of billionaires and their estimated net worth.

The dataset `mle/fp.dta` can be downloaded here or from its AER page.

In [3]:
```
pd.options.display.max_columns = 10

# Load in data and view
df = pd.read_stata('https://github.com/QuantEcon/lecture-source-
py/blob/master/source/_static/lecture_specific/mle/fp.dta?raw=true')
df.head()
```

Out[3]:
```
         country  ccode    year    cyear  numbil  …   topint08     rintr  \
0  United States    2.0  1990.0  21990.0     NaN  …  39.799999  4.988405
1  United States    2.0  1991.0  21991.0     NaN  …  39.799999  4.988405
2  United States    2.0  1992.0  21992.0     NaN  …  39.799999  4.988405
3  United States    2.0  1993.0  21993.0     NaN  …  39.799999  4.988405
4  United States    2.0  1994.0  21994.0     NaN  …  39.799999  4.988405

   noyrs  roflaw  nrrents
0   20.0    1.61      NaN
1   20.0    1.61      NaN
2   20.0    1.61      NaN
3   20.0    1.61      NaN
4   20.0    1.61      NaN

[5 rows x 36 columns]
```

Using a histogram, we can view the distribution of the number of billionaires per country, `numbil0`, in 2008 (the United States is dropped for plotting purposes)

In [4]:
```
numbil0_2008 = df[(df['year'] == 2008) & (
    df['country'] != 'United States')].loc[:, 'numbil0']

plt.subplots(figsize=(12, 8))
plt.hist(numbil0_2008, bins=30)
plt.xlim(left=0)
plt.grid()
plt.xlabel('Number of billionaires in 2008')
plt.ylabel('Count')
plt.show()
```

From the histogram, it appears that the Poisson assumption is not unreasonable (albeit with a very low μ and some outliers).

67.4 Conditional Distributions

In Treisman's paper, the dependent variable — the number of billionaires y_i in country i — is modeled as a function of GDP per capita, population size, and years membership in GATT and WTO.

Hence, the distribution of y_i needs to be conditioned on the vector of explanatory variables $\mathbf{x}_i$.

The standard formulation — the so-called *poisson regression* model — is as follows:

$$f(y_i \mid \mathbf{x}_i) = \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}; \qquad y_i = 0, 1, 2, \dots, \infty. \tag{1}$$

$$\text{where } \mu_i = \exp(\mathbf{x}_i' \beta) = \exp(\beta_0 + \beta_1 x_{i1} + \dots + \beta_k x_{ik})$$

To illustrate the idea that the distribution of y_i depends on $\mathbf{x}_i$ let's run a simple simulation.

We use our `poisson_pmf` function from above and arbitrary values for β and $\mathbf{x}_i$

```
In [5]: y_values = range(0, 20)

        # Define a parameter vector with estimates
        β = np.array([0.26, 0.18, 0.25, -0.1, -0.22])

        # Create some observations X
```

```
datasets = [np.array([0, 1, 1, 1, 2]),
            np.array([2, 3, 2, 4, 0]),
            np.array([3, 4, 5, 3, 2]),
            np.array([6, 5, 4, 4, 7])]


fig, ax = plt.subplots(figsize=(12, 8))

for X in datasets:
    μ = exp(X @ β)
    distribution = []
    for y_i in y_values:
        distribution.append(poisson_pmf(y_i, μ))
    ax.plot(y_values,
            distribution,
            label=f'$\mu_i$={μ:.1}',
            marker='o',
            markersize=8,
            alpha=0.5)

ax.grid()
ax.legend()
ax.set_xlabel('$y \mid x_i$')
ax.set_ylabel(r'$f(y \mid x_i; \beta )$')
ax.axis(xmin=0, ymin=0)
plt.show()
```

We can see that the distribution of y_i is conditional on $\mathbf{x}_i$ (μ_i is no longer constant).

67.5 Maximum Likelihood Estimation

In our model for number of billionaires, the conditional distribution contains 4 ($k = 4$) parameters that we need to estimate.

We will label our entire parameter vector as β where

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

To estimate the model using MLE, we want to maximize the likelihood that our estimate $\hat{\beta}$ is the true parameter β.

Intuitively, we want to find the $\hat{\beta}$ that best fits our data.

First, we need to construct the likelihood function $\mathcal{L}(\beta)$, which is similar to a joint probability density function.

Assume we have some data $y_i = \{y_1, y_2\}$ and $y_i \sim f(y_i)$.

If y_1 and y_2 are independent, the joint pmf of these data is $f(y_1, y_2) = f(y_1) \cdot f(y_2)$.

If y_i follows a Poisson distribution with $\lambda = 7$, we can visualize the joint pmf like so

```
In [6]: def plot_joint_poisson(μ=7, y_n=20):
            yi_values = np.arange(0, y_n, 1)

            # Create coordinate points of X and Y
            X, Y = np.meshgrid(yi_values, yi_values)

            # Multiply distributions together
            Z = poisson_pmf(X, μ) * poisson_pmf(Y, μ)

            fig = plt.figure(figsize=(12, 8))
            ax = fig.add_subplot(111, projection='3d')
            ax.plot_surface(X, Y, Z.T, cmap='terrain', alpha=0.6)
            ax.scatter(X, Y, Z.T, color='black', alpha=0.5, linewidths=1)
            ax.set(xlabel='$y_1$', ylabel='$y_2$')
            ax.set_zlabel('$f(y_1, y_2)$', labelpad=10)
            plt.show()

        plot_joint_poisson(μ=7, y_n=20)
```

Similarly, the joint pmf of our data (which is distributed as a conditional Poisson distribution) can be written as

$$f(y_1, y_2, \dots, y_n \mid \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n; \beta) = \prod_{i=1}^{n} \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}$$

y_i is conditional on both the values of $\mathbf{x}_i$ and the parameters β.

The likelihood function is the same as the joint pmf, but treats the parameter β as a random variable and takes the observations $(y_i, \mathbf{x}_i)$ as given

$$\begin{aligned}\mathcal{L}(\beta \mid y_1, y_2, \dots, y_n \ ; \ \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n) &= \prod_{i=1}^{n} \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i} \\ &= f(y_1, y_2, \dots, y_n \mid \ \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n; \beta)\end{aligned}$$

Now that we have our likelihood function, we want to find the $\hat{\beta}$ that yields the maximum likelihood value

$$\max_{\beta} \mathcal{L}(\beta)$$

In doing so it is generally easier to maximize the log-likelihood (consider differentiating $f(x) = x \exp(x)$ vs. $f(x) = \log(x) + x$).

Given that taking a logarithm is a monotone increasing transformation, a maximizer of the likelihood function will also be a maximizer of the log-likelihood function.

In our case the log-likelihood is

$$
\begin{aligned}
\log \mathcal{L}(\beta) = & \log \Big(f(y_1; \beta) \cdot f(y_2; \beta) \cdot \ldots \cdot f(y_n; \beta) \Big) \\
& = \sum_{i=1}^{n} \log f(y_i; \beta) \\
& = \sum_{i=1}^{n} \log \left(\frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i} \right) \\
& = \sum_{i=1}^{n} y_i \log \mu_i - \sum_{i=1}^{n} \mu_i - \sum_{i=1}^{n} \log y!
\end{aligned}
$$

The MLE of the Poisson to the Poisson for $\hat{\beta}$ can be obtained by solving

$$
\max_{\beta} \Big(\sum_{i=1}^{n} y_i \log \mu_i - \sum_{i=1}^{n} \mu_i - \sum_{i=1}^{n} \log y! \Big)
$$

However, no analytical solution exists to the above problem – to find the MLE we need to use numerical methods.

67.6 MLE with Numerical Methods

Many distributions do not have nice, analytical solutions and therefore require numerical methods to solve for parameter estimates.

One such numerical method is the Newton-Raphson algorithm.

Our goal is to find the maximum likelihood estimate $\hat{\beta}$.

At $\hat{\beta}$, the first derivative of the log-likelihood function will be equal to 0.

Let's illustrate this by supposing

$$
\log \mathcal{L}(\beta) = -(\beta - 10)^2 - 10
$$

In [7]:
```
β = np.linspace(1, 20)
logL = -(β - 10) ** 2 - 10
dlogL = -2 * β + 20

fig, (ax1, ax2) = plt.subplots(2, sharex=True, figsize=(12, 8))

ax1.plot(β, logL, lw=2)
ax2.plot(β, dlogL, lw=2)

ax1.set_ylabel(r'$log \mathcal{L(\beta)}$',
               rotation=0,
               labelpad=35,
               fontsize=15)
ax2.set_ylabel(r'$\frac{dlog \mathcal{L(\beta)}}{d \beta}$ ',
               rotation=0,
               labelpad=35,
               fontsize=19)
ax2.set_xlabel(r'$\beta$', fontsize=15)
```

```
ax1.grid(), ax2.grid()
plt.axhline(c='black')
plt.show()
```

The plot shows that the maximum likelihood value (the top plot) occurs when $\frac{d \log \mathcal{L}(\beta)}{d\beta} = 0$ (the bottom plot).

Therefore, the likelihood is maximized when $\beta = 10$.

We can also ensure that this value is a *maximum* (as opposed to a minimum) by checking that the second derivative (slope of the bottom plot) is negative.

The Newton-Raphson algorithm finds a point where the first derivative is 0.

To use the algorithm, we take an initial guess at the maximum value, β_0 (the OLS parameter estimates might be a reasonable guess), then

1. Use the updating rule to iterate the algorithm

$$\beta_{(k+1)} = \beta_{(k)} - H^{-1}(\beta_{(k)})G(\beta_{(k)})$$

where:

$$G(\beta_{(k)}) = \frac{d \log \mathcal{L}(\beta_{(k)})}{d\beta_{(k)}}$$

$$H(\beta_{(k)}) = \frac{d^2 \log \mathcal{L}(\beta_{(k)})}{d\beta_{(k)} d\beta'_{(k)}}$$

1. Check whether $\beta_{(k+1)} - \beta_{(k)} < tol$

- If true, then stop iterating and set $\hat{\beta} = \beta_{(k+1)}$

- If false, then update $\beta_{(k+1)}$

As can be seen from the updating equation, $\beta_{(k+1)} = \beta_{(k)}$ only when $G(\beta_{(k)}) = 0$ ie. where the first derivative is equal to 0.

(In practice, we stop iterating when the difference is below a small tolerance threshold)

Let's have a go at implementing the Newton-Raphson algorithm.

First, we'll create a class called `PoissonRegression` so we can easily recompute the values of the log likelihood, gradient and Hessian for every iteration

In [8]:
```
class PoissonRegression:

    def __init__(self, y, X, β):
        self.X = X
        self.n, self.k = X.shape
        # Reshape y as a n_by_1 column vector
        self.y = y.reshape(self.n,1)
        # Reshape β as a k_by_1 column vector
        self.β = β.reshape(self.k,1)

    def μ(self):
        return np.exp(self.X @ self.β)

    def logL(self):
        y = self.y
        μ = self.μ()
        return np.sum(y * np.log(μ) - μ - np.log(factorial(y)))

    def G(self):
        y = self.y
        μ = self.μ()
        return X.T @ (y - μ)

    def H(self):
        X = self.X
        μ = self.μ()
        return -(X.T @ (μ * X))
```

Our function `newton_raphson` will take a `PoissonRegression` object that has an initial guess of the parameter vector β_0.

The algorithm will update the parameter vector according to the updating rule, and recalculate the gradient and Hessian matrices at the new parameter estimates.

Iteration will end when either:

- The difference between the parameter and the updated parameter is below a tolerance level.
- The maximum number of iterations has been achieved (meaning convergence is not achieved).

So we can get an idea of what's going on while the algorithm is running, an option `display=True` is added to print out values at each iteration.

In [9]:
```
def newton_raphson(model, tol=1e-3, max_iter=1000, display=True):
```

```
    i = 0
    error = 100  # Initial error value

    # Print header of output
    if display:
        header = f'{"Iteration_k":<13}{"Log-likelihood":<16}{"θ":<60}'
        print(header)
        print("-" * len(header))

    # While loop runs while any value in error is greater
    # than the tolerance until max iterations are reached
    while np.any(error > tol) and i < max_iter:
        H, G = model.H(), model.G()
        β_new = model.β - (np.linalg.inv(H) @ G)
        error = β_new - model.β
        model.β = β_new

        # Print iterations
        if display:
            β_list = [f'{t:.3}' for t in list(model.β.flatten())]
            update = f'{i:<13}{model.logL():<16.8}{β_list}'
            print(update)

        i += 1

    print(f'Number of iterations: {i}')
    print(f'β_hat = {model.β.flatten()}')

    # Return a flat array for β (instead of a k_by_1 column vector)
    return model.β.flatten()
```

Let's try out our algorithm with a small dataset of 5 observations and 3 variables in $\mathbf{X}$.

```
In [10]: X = np.array([[1, 2, 5],
                       [1, 1, 3],
                       [1, 4, 2],
                       [1, 5, 2],
                       [1, 3, 1]])

         y = np.array([1, 0, 1, 1, 0])

         # Take a guess at initial βs
         init_β = np.array([0.1, 0.1, 0.1])

         # Create an object with Poisson model values
         poi = PoissonRegression(y, X, β=init_β)

         # Use newton_raphson to find the MLE
         β_hat = newton_raphson(poi, display=True)
```

```
    Iteration_k  Log-likelihood  θ
-------------------------------------------------------------------------------
---
---
0            -4.3447622      ['-1.49', '0.265', '0.244']
1            -3.5742413      ['-3.38', '0.528', '0.474']
2            -3.3999526      ['-5.06', '0.782', '0.702']
```

```
3            -3.3788646      ['-5.92', '0.909', '0.82']
4            -3.3783559      ['-6.07', '0.933', '0.843']
5            -3.3783555      ['-6.08', '0.933', '0.843']
Number of iterations: 6
β_hat = [-6.07848205  0.93340226  0.84329625]
```

As this was a simple model with few observations, the algorithm achieved convergence in only 6 iterations.

You can see that with each iteration, the log-likelihood value increased.

Remember, our objective was to maximize the log-likelihood function, which the algorithm has worked to achieve.

Also, note that the increase in $\log \mathcal{L}(\beta_{(k)})$ becomes smaller with each iteration.

This is because the gradient is approaching 0 as we reach the maximum, and therefore the numerator in our updating equation is becoming smaller.

The gradient vector should be close to 0 at $\hat{\beta}$

In [11]:
```
poi.G()
```

Out[11]:
```
array([[-3.95169226e-07],
       [-1.00114804e-06],
       [-7.73114559e-07]])
```

The iterative process can be visualized in the following diagram, where the maximum is found at $\beta = 10$

In [12]:
```
logL = lambda x: -(x - 10) ** 2 - 10

def find_tangent(β, a=0.01):
    y1 = logL(β)
    y2 = logL(β+a)
    x = np.array([[β, 1], [β+a, 1]])
    m, c = np.linalg.lstsq(x, np.array([y1, y2]), rcond=None)[0]
    return m, c

β = np.linspace(2, 18)
fig, ax = plt.subplots(figsize=(12, 8))
ax.plot(β, logL(β), lw=2, c='black')

for β in [7, 8.5, 9.5, 10]:
    β_line = np.linspace(β-2, β+2)
    m, c = find_tangent(β)
    y = m * β_line + c
    ax.plot(β_line, y, '-', c='purple', alpha=0.8)
    ax.text(β+2.05, y[-1], f'$G({β}) = {abs(m):.0f}$', fontsize=12)
    ax.vlines(β, -24, logL(β), linestyles='--', alpha=0.5)
    ax.hlines(logL(β), 6, β, linestyles='--', alpha=0.5)

ax.set(ylim=(-24, -4), xlim=(6, 13))
ax.set_xlabel(r'$\beta$', fontsize=15)
ax.set_ylabel(r'$log \mathcal{L(\beta)}$',
              rotation=0,
```

```
                 labelpad=25,
                 fontsize=15)
ax.grid(alpha=0.3)
plt.show()
```

Note that our implementation of the Newton-Raphson algorithm is rather basic — for more robust implementations see, for example, scipy.optimize.

67.7 Maximum Likelihood Estimation with `statsmodels`

Now that we know what's going on under the hood, we can apply MLE to an interesting application.

We'll use the Poisson regression model in `statsmodels` to obtain a richer output with standard errors, test values, and more.

`statsmodels` uses the same algorithm as above to find the maximum likelihood estimates.

Before we begin, let's re-estimate our simple model with `statsmodels` to confirm we obtain the same coefficients and log-likelihood value.

```
In [13]: X = np.array([[1, 2, 5],
                       [1, 1, 3],
                       [1, 4, 2],
                       [1, 5, 2],
                       [1, 3, 1]])

         y = np.array([1, 0, 1, 1, 0])

         stats_poisson = Poisson(y, X).fit()
         print(stats_poisson.summary())
```

```
    Optimization terminated successfully.
         Current function value: 0.675671
         Iterations 7
                          Poisson Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                    5
Model:                        Poisson   Df Residuals:                        2
Method:                           MLE   Df Model:                            2
Date:                Tue, 03 Mar 2020   Pseudo R-squ.:                  0.2546
Time:                        17:06:56   Log-Likelihood:                -3.3784
converged:                       True   LL-Null:                       -4.5325
Covariance Type:            nonrobust   LLR p-value:                    0.3153
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -6.0785      5.279     -1.151      0.250     -16.425       4.268
x1             0.9334      0.829      1.126      0.260      -0.691       2.558
x2             0.8433      0.798      1.057      0.291      -0.720       2.407
==============================================================================
```

Now let's replicate results from Daniel Treisman's paper, Russia's Billionaires, mentioned earlier in the lecture.

Treisman starts by estimating equation (1), where:

- y_i is $number\ of\ billionaires_i$
- x_{i1} is $\log GDP\ per\ capita_i$
- x_{i2} is $\log population_i$
- x_{i3} is $years\ in\ GATT_i$ – years membership in GATT and WTO (to proxy access to international markets)

The paper only considers the year 2008 for estimation.

We will set up our variables for estimation like so (you should have the data assigned to `df` from earlier in the lecture)

In [14]:
```
# Keep only year 2008
df = df[df['year'] == 2008]

# Add a constant
df['const'] = 1

# Variable sets
reg1 = ['const', 'lngdppc', 'lnpop', 'gattwto08']
reg2 = ['const', 'lngdppc', 'lnpop',
        'gattwto08', 'lnmcap08', 'rintr', 'topint08']
reg3 = ['const', 'lngdppc', 'lnpop', 'gattwto08', 'lnmcap08',
        'rintr', 'topint08', 'nrrents', 'roflaw']
```

Then we can use the `Poisson` function from `statsmodels` to fit the model.

We'll use robust standard errors as in the author's paper

In [15]:
```
# Specify model
poisson_reg = sm.Poisson(df[['numbil0']], df[reg1],
                         missing='drop').fit(cov_type='HC0')
print(poisson_reg.summary())
```

```
    Optimization terminated successfully.
         Current function value: 2.226090
         Iterations 9
                          Poisson Regression Results
==============================================================================
Dep. Variable:                numbil0   No. Observations:                  197
Model:                        Poisson   Df Residuals:                      193
Method:                           MLE   Df Model:                            3
Date:                Tue, 03 Mar 2020   Pseudo R-squ.:                  0.8574
Time:                        17:06:56   Log-Likelihood:                -438.54
converged:                       True   LL-Null:                       -3074.7
Covariance Type:                  HC0   LLR p-value:                     0.000
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const        -29.0495      2.578    -11.268      0.000     -34.103     -23.997
lngdppc        1.0839      0.138      7.834      0.000       0.813       1.355
lnpop          1.1714      0.097     12.024      0.000       0.980       1.362
gattwto08      0.0060      0.007      0.868      0.386      -0.008       0.019
==============================================================================
```

Success! The algorithm was able to achieve convergence in 9 iterations.

Our output indicates that GDP per capita, population, and years of membership in the General Agreement on Tariffs and Trade (GATT) are positively related to the number of billionaires a country has, as expected.

Let's also estimate the author's more full-featured models and display them in a single table

In [16]:
```
regs = [reg1, reg2, reg3]
reg_names = ['Model 1', 'Model 2', 'Model 3']
info_dict = {'Pseudo R-squared': lambda x: f"{x.prsquared:.2f}",
             'No. observations': lambda x: f"{int(x.nobs):d}"}
regressor_order = ['const',
                   'lngdppc',
                   'lnpop',
                   'gattwto08',
                   'lnmcap08',
                   'rintr',
                   'topint08',
                   'nrrents',
                   'roflaw']
results = []

for reg in regs:
    result = sm.Poisson(df[['numbil0']], df[reg],
                        missing='drop').fit(cov_type='HC0',
                                            maxiter=100, disp=0)
    results.append(result)

results_table = summary_col(results=results,
                            float_format='%0.3f',
                            stars=True,
                            model_names=reg_names,
                            info_dict=info_dict,
                            regressor_order=regressor_order)
results_table.add_title('Table 1 - Explaining the Number of Billionaires \
                        in 2008')
print(results_table)
```

```
    Table 1 - Explaining the Number of Billionaires                         in 2008
==================================================
                   Model 1    Model 2    Model 3
--------------------------------------------------
const            -29.050*** -19.444*** -20.858***
                 (2.578)    (4.820)    (4.255)
lngdppc          1.084***   0.717***   0.737***
                 (0.138)    (0.244)    (0.233)
lnpop            1.171***   0.806***   0.929***
                 (0.097)    (0.213)    (0.195)
gattwto08        0.006      0.007      0.004
                 (0.007)    (0.006)    (0.006)
lnmcap08                    0.399**    0.286*
                            (0.172)    (0.167)
rintr                       -0.010     -0.009
                            (0.010)    (0.010)
topint08                    -0.051***  -0.058***
                            (0.011)    (0.012)
nrrents                                -0.005
                                       (0.010)
roflaw                                 0.203
                                       (0.372)
Pseudo R-squared 0.86       0.90       0.90
No. observations 197        131        131
==================================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
```

The output suggests that the frequency of billionaires is positively correlated with GDP per capita, population size, stock market capitalization, and negatively correlated with top marginal income tax rate.

To analyze our results by country, we can plot the difference between the predicted an actual values, then sort from highest to lowest and plot the first 15

```
In [17]: data = ['const', 'lngdppc', 'lnpop', 'gattwto08', 'lnmcap08', 'rintr',
                 'topint08', 'nrrents', 'roflaw', 'numbil0', 'country']
         results_df = df[data].dropna()

         # Use last model (model 3)
         results_df['prediction'] = results[-1].predict()

         # Calculate difference
        results_df['difference'] = results_df['numbil0'] - results_df['prediction']

         # Sort in descending order
         results_df.sort_values('difference', ascending=False, inplace=True)

         # Plot the first 15 data points
         results_df[:15].plot('country', 'difference', kind='bar',
                              figsize=(12,8), legend=False)
         plt.ylabel('Number of billionaires above predicted level')
         plt.xlabel('Country')
         plt.show()
```

As we can see, Russia has by far the highest number of billionaires in excess of what is predicted by the model (around 50 more than expected).

Treisman uses this empirical result to discuss possible reasons for Russia's excess of billionaires, including the origination of wealth in Russia, the political climate, and the history of privatization in the years after the USSR.

67.8 Summary

In this lecture, we used Maximum Likelihood Estimation to estimate the parameters of a Poisson model.

`statsmodels` contains other built-in likelihood models such as Probit and Logit.

For further flexibility, `statsmodels` provides a way to specify the distribution manually using the `GenericLikelihoodModel` class - an example notebook can be found here.

67.9 Exercises

67.9.1 Exercise 1

Suppose we wanted to estimate the probability of an event y_i occurring, given some observations.

We could use a probit regression model, where the pmf of y_i is

$$f(y_i;\beta) = \mu_i^{y_i}(1-\mu_i)^{1-y_i}, \quad y_i = 0,1$$
$$\text{where} \quad \mu_i = \Phi(\mathbf{x}_i'\beta)$$

Φ represents the *cumulative normal distribution* and constrains the predicted y_i to be between 0 and 1 (as required for a probability).

β is a vector of coefficients.

Following the example in the lecture, write a class to represent the Probit model.

To begin, find the log-likelihood function and derive the gradient and Hessian.

The `scipy` module `stats.norm` contains the functions needed to compute the cmf and pmf of the normal distribution.

67.9.2 Exercise 2

Use the following dataset and initial values of β to estimate the MLE with the Newton-Raphson algorithm developed earlier in the lecture

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 4 \\ 1 & 1 & 1 \\ 1 & 4 & 3 \\ 1 & 5 & 6 \\ 1 & 3 & 5 \end{bmatrix} \quad y = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \beta_{(0)} = \begin{bmatrix} 0.1 \\ 0.1 \\ 0.1 \end{bmatrix}$$

Verify your results with `statsmodels` - you can import the Probit function with the following import statement

```
In [18]: from statsmodels.discrete.discrete_model import Probit
```

Note that the simple Newton-Raphson algorithm developed in this lecture is very sensitive to initial values, and therefore you may fail to achieve convergence with different starting values.

67.10 Solutions

67.10.1 Exercise 1

The log-likelihood can be written as

$$\log \mathcal{L} = \sum_{i=1}^{n} \left[y_i \log \Phi(\mathbf{x}_i'\beta) + (1-y_i)\log(1-\Phi(\mathbf{x}_i'\beta)) \right]$$

Using the **fundamental theorem of calculus**, the derivative of a cumulative probability distribution is its marginal distribution

$$\frac{\partial}{\partial s}\Phi(s) = \phi(s)$$

where ϕ is the marginal normal distribution.

The gradient vector of the Probit model is

$$\frac{\partial \log \mathcal{L}}{\partial \beta} = \sum_{i=1}^{n} \Big[y_i \frac{\phi(\mathbf{x}_i'\beta)}{\Phi(\mathbf{x}_i'\beta)} - (1 - y_i) \frac{\phi(\mathbf{x}_i'\beta)}{1 - \Phi(\mathbf{x}_i'\beta)} \Big] \mathbf{x}_i$$

The Hessian of the Probit model is

$$\frac{\partial^2 \log \mathcal{L}}{\partial \beta \partial \beta'} = -\sum_{i=1}^{n} \phi(\mathbf{x}_i'\beta) \Big[y_i \frac{\phi(\mathbf{x}_i'\beta) + \mathbf{x}_i'\beta\Phi(\mathbf{x}_i'\beta)}{[\Phi(\mathbf{x}_i'\beta)]^2} + (1 - y_i) \frac{\phi_i(\mathbf{x}_i'\beta) - \mathbf{x}_i'\beta(1 - \Phi(\mathbf{x}_i'\beta))}{[1 - \Phi(\mathbf{x}_i'\beta)]^2} \Big] \mathbf{x}_i \mathbf{x}_i'$$

Using these results, we can write a class for the Probit model as follows

```
In [19]: class ProbitRegression:

             def __init__(self, y, X, β):
                 self.X, self.y, self.β = X, y, β
                 self.n, self.k = X.shape

             def μ(self):
                 return norm.cdf(self.X @ self.β.T)

             def ϕ(self):
                 return norm.pdf(self.X @ self.β.T)

             def logL(self):
                 μ = self.μ()
                 return np.sum(y * np.log(μ) + (1 - y) * np.log(1 - μ))

             def G(self):
                 μ = self.μ()
                 ϕ = self.ϕ()
                 return np.sum((X.T * y * ϕ / μ - X.T * (1 - y) * ϕ / (1 - μ)),
                               axis=1)

             def H(self):
                 X = self.X
                 β = self.β
                 μ = self.μ()
                 ϕ = self.ϕ()
                 a = (ϕ + (X @ β.T) * μ) / μ**2
                 b = (ϕ - (X @ β.T) * (1 - μ)) / (1 - μ)**2
                 return -(ϕ * (y * a + (1 - y) * b) * X.T) @ X
```

67.10.2 Exercise 2

```
In [20]: X = np.array([[1, 2, 4],
                       [1, 1, 1],
                       [1, 4, 3],
                       [1, 5, 6],
                       [1, 3, 5]])

         y = np.array([1, 0, 1, 1, 0])
```

```
# Take a guess at initial βs
β = np.array([0.1, 0.1, 0.1])

# Create instance of Probit regression class
prob = ProbitRegression(y, X, β)

# Run Newton-Raphson algorithm
newton_raphson(prob)
```

```
  Iteration_k  Log-likelihood  θ
--------------------------------------------------------------------------------
---
---
0            -2.3796884      ['-1.34', '0.775', '-0.157']
1            -2.3687526      ['-1.53', '0.775', '-0.0981']
2            -2.3687294      ['-1.55', '0.778', '-0.0971']
3            -2.3687294      ['-1.55', '0.778', '-0.0971']
Number of iterations: 4
β_hat = [-1.54625858  0.77778952 -0.09709757]
```

Out[20]: array([-1.54625858, 0.77778952, -0.09709757])

In [21]:
```
# Use statsmodels to verify results

print(Probit(y, X).fit().summary())
```

```
  Optimization terminated successfully.
       Current function value: 0.473746
       Iterations 6
                           Probit Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                    5
Model:                         Probit   Df Residuals:                        2
Method:                           MLE   Df Model:                            2
Date:                Tue, 03 Mar 2020   Pseudo R-squ.:                  0.2961
Time:                        17:06:57   Log-Likelihood:                -2.3687
converged:                       True   LL-Null:                       -3.3651
Covariance Type:            nonrobust   LLR p-value:                    0.3692
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -1.5463      1.866     -0.829      0.407      -5.204       2.111
x1             0.7778      0.788      0.986      0.324      -0.768       2.323
x2            -0.0971      0.590     -0.165      0.869      -1.254       1.060
==============================================================================
```

Part IX

Multiple Agent Models

Chapter 68

Schelling's Segregation Model

68.1 Contents

- Outline 68.2
- The Model 68.3
- Results 68.4
- Exercises 68.5
- Solutions 68.6

68.2 Outline

In 1969, Thomas C. Schelling developed a simple but striking model of racial segregation [145].

His model studies the dynamics of racially mixed neighborhoods.

Like much of Schelling's work, the model shows how local interactions can lead to surprising aggregate structure.

In particular, it shows that relatively mild preference for neighbors of similar race can lead in aggregate to the collapse of mixed neighborhoods, and high levels of segregation.

In recognition of this and other research, Schelling was awarded the 2005 Nobel Prize in Economic Sciences (joint with Robert Aumann).

In this lecture, we (in fact you) will build and run a version of Schelling's model.

Let's start with some imports:

In [1]:
```
from random import uniform, seed
from math import sqrt
import matplotlib.pyplot as plt
%matplotlib inline
```

68.3 The Model

We will cover a variation of Schelling's model that is easy to program and captures the main idea.

68.3.1 Set-Up

Suppose we have two types of people: orange people and green people.

For the purpose of this lecture, we will assume there are 250 of each type.

These agents all live on a single unit square.

The location of an agent is just a point (x, y), where $0 < x, y < 1$.

68.3.2 Preferences

We will say that an agent is *happy* if half or more of her 10 nearest neighbors are of the same type.

Here 'nearest' is in terms of Euclidean distance.

An agent who is not happy is called *unhappy*.

An important point here is that agents are not averse to living in mixed areas.

They are perfectly happy if half their neighbors are of the other color.

68.3.3 Behavior

Initially, agents are mixed together (integrated).

In particular, the initial location of each agent is an independent draw from a bivariate uniform distribution on $S = (0, 1)^2$.

Now, cycling through the set of all agents, each agent is now given the chance to stay or move.

We assume that each agent will stay put if they are happy and move if unhappy.

The algorithm for moving is as follows

1. Draw a random location in S
2. If happy at new location, move there
3. Else, go to step 1

In this way, we cycle continuously through the agents, moving as required.

We continue to cycle until no one wishes to move.

68.4 Results

Let's have a look at the results we got when we coded and ran this model.

As discussed above, agents are initially mixed randomly together.

But after several cycles, they become segregated into distinct regions.

Cycle 1
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

Cycle 2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

In this instance, the program terminated after 4 cycles through the set of agents, indicating that all agents had reached a state of happiness.

What is striking about the pictures is how rapidly racial integration breaks down.

This is despite the fact that people in the model don't actually mind living mixed with the other type.

Even with these preferences, the outcome is a high degree of segregation.

68.5 Exercises

68.5.1 Exercise 1

Implement and run this simulation for yourself.

Consider the following structure for your program.

Agents can be modeled as objects.

Here's an indication of how they might look

```
* Data:
```

```
    * type (green or orange)
    * location

* Methods:

    * determine whether happy or not given locations of other agents

    * If not happy, move

        * find a new location where happy
```

And here's some pseudocode for the main loop

```
while agents are still moving
    for agent in agents
        give agent the opportunity to move
```

Use 250 agents of each type.

68.6 Solutions

68.6.1 Exercise 1

Here's one solution that does the job we want.

If you feel like a further exercise, you can probably speed up some of the computations and then increase the number of agents.

```
In [2]: seed(10)  # For reproducible random numbers

        class Agent:

            def __init__(self, type):
                self.type = type
                self.draw_location()

            def draw_location(self):
                self.location = uniform(0, 1), uniform(0, 1)

            def get_distance(self, other):
                "Computes the euclidean distance between self and other agent."
                a = (self.location[0] - other.location[0])**2
                b = (self.location[1] - other.location[1])**2
                return sqrt(a + b)

            def happy(self, agents):
                "True if sufficient number of nearest neighbors are of the same
 ↪type."
                distances = []
                # distances is a list of pairs (d, agent), where d is distance from
                # agent to self
                for agent in agents:
```

```
                if self != agent:
                    distance = self.get_distance(agent)
                    distances.append((distance, agent))
            # == Sort from smallest to largest, according to distance == #
            distances.sort()
            # == Extract the neighboring agents == #
            neighbors = [agent for d, agent in distances[:num_neighbors]]
            # == Count how many neighbors have the same type as self == #
            num_same_type = sum(self.type == agent.type for agent in neighbors)
            return num_same_type >= require_same_type

        def update(self, agents):
            "If not happy, then randomly choose new locations until happy."
            while not self.happy(agents):
                self.draw_location()


    def plot_distribution(agents, cycle_num):
        "Plot the distribution of agents after cycle_num rounds of the loop."
        x_values_0, y_values_0 = [], []
        x_values_1, y_values_1 = [], []
        # == Obtain locations of each type == #
        for agent in agents:
            x, y = agent.location
            if agent.type == 0:
                x_values_0.append(x)
                y_values_0.append(y)
            else:
                x_values_1.append(x)
                y_values_1.append(y)
        fig, ax = plt.subplots(figsize=(8, 8))
        plot_args = {'markersize': 8, 'alpha': 0.6}
        ax.set_facecolor('azure')
        ax.plot(x_values_0, y_values_0, 'o', markerfacecolor='orange',
↪**plot_args)
        ax.plot(x_values_1, y_values_1, 'o', markerfacecolor='green',
↪**plot_args)
        ax.set_title(f'Cycle {cycle_num-1}')
        plt.show()

    # == Main == #

    num_of_type_0 = 250
    num_of_type_1 = 250
    num_neighbors = 10      # Number of agents regarded as neighbors
    require_same_type = 5   # Want at least this many neighbors to be same type

    # == Create a list of agents == #
    agents = [Agent(0) for i in range(num_of_type_0)]
    agents.extend(Agent(1) for i in range(num_of_type_1))


    count = 1
    # ==  Loop until none wishes to move == #
    while True:
        print('Entering loop ', count)
        plot_distribution(agents, count)
        count += 1
```

```
    no_one_moved = True
    for agent in agents:
        old_location = agent.location
        agent.update(agents)
        if agent.location != old_location:
            no_one_moved = False
    if no_one_moved:
        break

print('Converged, terminating.')
```

```
Entering loop  1
```

```
Entering loop  2
```

```
Entering loop  3
```

```
Entering loop  4
```

```
Converged, terminating.
```

Chapter 69

A Lake Model of Employment and Unemployment

69.1 Contents

- Overview 69.2
- The Model 69.3
- Implementation 69.4
- Dynamics of an Individual Worker 69.5
- Endogenous Job Finding Rate 69.6
- Exercises 69.7
- Solutions 69.8
- Lake Model Solutions 69.9

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

69.2 Overview

This lecture describes what has come to be called a *lake model*.

The lake model is a basic tool for modeling unemployment.

It allows us to analyze

- flows between unemployment and employment.
- how these flows influence steady state employment and unemployment rates.

It is a good model for interpreting monthly labor department reports on gross and net jobs created and jobs destroyed.

The "lakes" in the model are the pools of employed and unemployed.

The "flows" between the lakes are caused by

- firing and hiring
- entry and exit from the labor force

For the first part of this lecture, the parameters governing transitions into and out of unem-

ployment and employment are exogenous.

Later, we'll determine some of these transition rates endogenously using the McCall search model.

We'll also use some nifty concepts like ergodicity, which provides a fundamental link between *cross-sectional* and *long run time series* distributions.

These concepts will help us build an equilibrium model of ex-ante homogeneous workers whose different luck generates variations in their ex post experiences.

Let's start with some imports:

In [2]:
```
import numpy as np
from quantecon import MarkovChain
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.stats import norm
from scipy.optimize import brentq
from quantecon.distributions import BetaBinomial
from numba import jit
```

69.2.1 Prerequisites

Before working through what follows, we recommend you read the lecture on finite Markov chains.

You will also need some basic linear algebra and probability.

69.3 The Model

The economy is inhabited by a very large number of ex-ante identical workers.

The workers live forever, spending their lives moving between unemployment and employment.

Their rates of transition between employment and unemployment are governed by the following parameters:

- λ, the job finding rate for currently unemployed workers
- α, the dismissal rate for currently employed workers
- b, the entry rate into the labor force
- d, the exit rate from the labor force

The growth rate of the labor force evidently equals $g = b - d$.

69.3.1 Aggregate Variables

We want to derive the dynamics of the following aggregates

- E_t, the total number of employed workers at date t
- U_t, the total number of unemployed workers at t
- N_t, the number of workers in the labor force at t

We also want to know the values of the following objects

- The employment rate $e_t := E_t/N_t$.
- The unemployment rate $u_t := U_t/N_t$.

(Here and below, capital letters represent stocks and lowercase letters represent flows)

69.3.2 Laws of Motion for Stock Variables

We begin by constructing laws of motion for the aggregate variables E_t, U_t, N_t.

Of the mass of workers E_t who are employed at date t,

- $(1-d)E_t$ will remain in the labor force
- of these, $(1-\alpha)(1-d)E_t$ will remain employed

Of the mass of workers U_t workers who are currently unemployed,

- $(1-d)U_t$ will remain in the labor force
- of these, $(1-d)\lambda U_t$ will become employed

Therefore, the number of workers who will be employed at date $t+1$ will be

$$E_{t+1} = (1-d)(1-\alpha)E_t + (1-d)\lambda U_t$$

A similar analysis implies

$$U_{t+1} = (1-d)\alpha E_t + (1-d)(1-\lambda)U_t + b(E_t + U_t)$$

The value $b(E_t + U_t)$ is the mass of new workers entering the labor force unemployed.

The total stock of workers $N_t = E_t + U_t$ evolves as

$$N_{t+1} = (1+b-d)N_t = (1+g)N_t$$

Letting $X_t := \begin{pmatrix} U_t \\ E_t \end{pmatrix}$, the law of motion for X is

$$X_{t+1} = AX_t \quad \text{where} \quad A := \begin{pmatrix} (1-d)(1-\lambda)+b & (1-d)\alpha + b \\ (1-d)\lambda & (1-d)(1-\alpha) \end{pmatrix}$$

This law tells us how total employment and unemployment evolve over time.

69.3.3 Laws of Motion for Rates

Now let's derive the law of motion for rates.

To get these we can divide both sides of $X_{t+1} = AX_t$ by N_{t+1} to get

$$\begin{pmatrix} U_{t+1}/N_{t+1} \\ E_{t+1}/N_{t+1} \end{pmatrix} = \frac{1}{1+g} A \begin{pmatrix} U_t/N_t \\ E_t/N_t \end{pmatrix}$$

Letting

$$x_t := \begin{pmatrix} u_t \\ e_t \end{pmatrix} = \begin{pmatrix} U_t/N_t \\ E_t/N_t \end{pmatrix}$$

we can also write this as

$$x_{t+1} = \hat{A} x_t \quad \text{where} \quad \hat{A} := \frac{1}{1+g} A$$

You can check that $e_t + u_t = 1$ implies that $e_{t+1} + u_{t+1} = 1$.

This follows from the fact that the columns of $\hat{A}$ sum to 1.

69.4 Implementation

Let's code up these equations.

To do this we're going to use a class that we'll call LakeModel.

This class will

1. store the primitives α, λ, b, d
2. compute and store the implied objects $g, A, \hat{A}$
3. provide methods to simulate dynamics of the stocks and rates
4. provide a method to compute the state state of the rate

To write a nice implementation, there's an issue we have to address.

Derived data such as A depend on the primitives like α and λ.

If a user alters these primitives, we would ideally like derived data to update automatically.

(For example, if a user changes the value of b for a given instance of the class, we would like $g = b - d$ to update automatically)

To achieve this outcome, we're going to use descriptors and decorators such as @property.

If you need to refresh your understanding of how these work, consult this lecture.

Here's the code:

In [3]:
```
class LakeModel:
    """
    Solves the lake model and computes dynamics of unemployment stocks and
    rates.

    Parameters:
    ------------
    λ : scalar
        The job finding rate for currently unemployed workers
    α : scalar
        The dismissal rate for currently employed workers
    b : scalar
        Entry rate into the labor force
    d : scalar
        Exit rate from the labor force

    """
```

```
    def __init__(self, λ=0.283, α=0.013, b=0.0124, d=0.00822):
        self._λ, self._α, self._b, self._d = λ, α, b, d
        self.compute_derived_values()

    def compute_derived_values(self):
        # Unpack names to simplify expression
        λ, α, b, d = self._λ, self._α, self._b, self._d

        self._g = b - d
        self._A = np.array([[(1-d) * (1-λ) + b,      (1 - d) * α + b],
                            [        (1-d) * λ,   (1 - d) * (1 - α)]])

        self._A_hat = self._A / (1 + self._g)

    @property
    def g(self):
        return self._g

    @property
    def A(self):
        return self._A

    @property
    def A_hat(self):
        return self._A_hat

    @property
    def λ(self):
        return self._λ

    @λ.setter
    def λ(self, new_value):
        self._α = new_value
        self.compute_derived_values()

    @property
    def α(self):
        return self._α

    @α.setter
    def α(self, new_value):
        self._α = new_value
        self.compute_derived_values()

    @property
    def b(self):
        return self._b

    @b.setter
    def b(self, new_value):
        self._b = new_value
        self.compute_derived_values()

    @property
    def d(self):
        return self._d

    @d.setter
```

```
    def d(self, new_value):
        self._d = new_value
        self.compute_derived_values()


    def rate_steady_state(self, tol=1e-6):
        """
        Finds the steady state of the system :math:`x_{t+1} = \hat A x_{t}`

        Returns
        --------
        xbar : steady state vector of employment and unemployment rates
        """
        x = 0.5 * np.ones(2)
        error = tol + 1
        while error > tol:
            new_x = self.A_hat @ x
            error = np.max(np.abs(new_x - x))
            x = new_x
        return x

    def simulate_stock_path(self, X0, T):
        """
        Simulates the sequence of Employment and Unemployment stocks

        Parameters
        ------------
        X0 : array
            Contains initial values (E0, U0)
        T : int
            Number of periods to simulate

        Returns
        ---------
        X : iterator
            Contains sequence of employment and unemployment stocks
        """

        X = np.atleast_1d(X0)  # Recast as array just in case
        for t in range(T):
            yield X
            X = self.A @ X

    def simulate_rate_path(self, x0, T):
        """
        Simulates the sequence of employment and unemployment rates

        Parameters
        ------------
        x0 : array
            Contains initial values (e0,u0)
        T : int
            Number of periods to simulate

        Returns
        ---------
        x : iterator
            Contains sequence of employment and unemployment rates
```

```
        """
        x = np.atleast_1d(x0)  # Recast as array just in case
        for t in range(T):
            yield x
            x = self.A_hat @ x
```

As desired, if we create an instance and update a primitive like α, derived objects like A will also change

```
In [4]: lm = LakeModel()
        lm.α
```

```
Out[4]: 0.013
```

```
In [5]: lm.A
```

```
Out[5]: array([[0.72350626, 0.02529314],
               [0.28067374, 0.97888686]])
```

```
In [6]: lm.α = 2
        lm.A
```

```
Out[6]: array([[ 0.72350626,  1.99596   ],
               [ 0.28067374, -0.99178   ]])
```

69.4.1 Aggregate Dynamics

Let's run a simulation under the default parameters (see above) starting from $X_0 = (12, 138)$

```
In [7]: lm = LakeModel()
        N_0 = 150      # Population
        e_0 = 0.92     # Initial employment rate
        u_0 = 1 - e_0  # Initial unemployment rate
        T = 50         # Simulation length

        U_0 = u_0 * N_0
        E_0 = e_0 * N_0

        fig, axes = plt.subplots(3, 1, figsize=(10, 8))
        X_0 = (U_0, E_0)
        X_path = np.vstack(tuple(lm.simulate_stock_path(X_0, T)))

        axes[0].plot(X_path[:, 0], lw=2)
        axes[0].set_title('Unemployment')

        axes[1].plot(X_path[:, 1], lw=2)
        axes[1].set_title('Employment')

        axes[2].plot(X_path.sum(1), lw=2)
        axes[2].set_title('Labor force')

        for ax in axes:
```

```
    ax.grid()

plt.tight_layout()
plt.show()
```

The aggregates E_t and U_t don't converge because their sum $E_t + U_t$ grows at rate g.

On the other hand, the vector of employment and unemployment rates x_t can be in a steady state $\bar{x}$ if there exists an $\bar{x}$ such that

- $\bar{x} = \hat{A}\bar{x}$
- the components satisfy $\bar{e} + \bar{u} = 1$

This equation tells us that a steady state level $\bar{x}$ is an eigenvector of $\hat{A}$ associated with a unit eigenvalue.

We also have $x_t \to \bar{x}$ as $t \to \infty$ provided that the remaining eigenvalue of $\hat{A}$ has modulus less that 1.

This is the case for our default parameters:

```
In [8]: lm = LakeModel()
        e, f = np.linalg.eigvals(lm.A_hat)
        abs(e), abs(f)
```

```
Out[8]: (0.6953067378358462, 1.0)
```

Let's look at the convergence of the unemployment and employment rate to steady state levels (dashed red line)

```
In [9]: lm = LakeModel()
        e_0 = 0.92     # Initial employment rate
        u_0 = 1 - e_0  # Initial unemployment rate
        T = 50         # Simulation length

        xbar = lm.rate_steady_state()

        fig, axes = plt.subplots(2, 1, figsize=(10, 8))
        x_0 = (u_0, e_0)
        x_path = np.vstack(tuple(lm.simulate_rate_path(x_0, T)))

        titles = ['Unemployment rate', 'Employment rate']

        for i, title in enumerate(titles):
            axes[i].plot(x_path[:, i], lw=2, alpha=0.5)
            axes[i].hlines(xbar[i], 0, T, 'r', '--')
            axes[i].set_title(title)
            axes[i].grid()

        plt.tight_layout()
        plt.show()
```

69.5 Dynamics of an Individual Worker

An individual worker's employment dynamics are governed by a finite state Markov process.

The worker can be in one of two states:

- $s_t = 0$ means unemployed
- $s_t = 1$ means employed

Let's start off under the assumption that $b = d = 0$.

The associated transition matrix is then

$$P = \begin{pmatrix} 1 - \lambda & \lambda \\ \alpha & 1 - \alpha \end{pmatrix}$$

Let ψ_t denote the marginal distribution over employment/unemployment states for the worker at time t.

As usual, we regard it as a row vector.

We know from an earlier discussion that ψ_t follows the law of motion

$$\psi_{t+1} = \psi_t P$$

We also know from the lecture on finite Markov chains that if $\alpha \in (0, 1)$ and $\lambda \in (0, 1)$, then P has a unique stationary distribution, denoted here by ψ^*.

The unique stationary distribution satisfies

$$\psi^*[0] = \frac{\alpha}{\alpha + \lambda}$$

Not surprisingly, probability mass on the unemployment state increases with the dismissal rate and falls with the job finding rate.

69.5.1 Ergodicity

Let's look at a typical lifetime of employment-unemployment spells.

We want to compute the average amounts of time an infinitely lived worker would spend employed and unemployed.

Let

$$\bar{s}_{u,T} := \frac{1}{T} \sum_{t=1}^{T} \mathbb{1}\{s_t = 0\}$$

and

$$\bar{s}_{e,T} := \frac{1}{T} \sum_{t=1}^{T} \mathbb{1}\{s_t = 1\}$$

(As usual, $\mathbb{1}\{Q\} = 1$ if statement Q is true and 0 otherwise)

These are the fraction of time a worker spends unemployed and employed, respectively, up until period T.

If $\alpha \in (0, 1)$ and $\lambda \in (0, 1)$, then P is ergodic, and hence we have

$$\lim_{T\to\infty} \bar{s}_{u,T} = \psi^*[0] \quad \text{and} \quad \lim_{T\to\infty} \bar{s}_{e,T} = \psi^*[1]$$

with probability one.

Inspection tells us that P is exactly the transpose of $\hat{A}$ under the assumption $b = d = 0$.

Thus, the percentages of time that an infinitely lived worker spends employed and unemployed equal the fractions of workers employed and unemployed in the steady state distribution.

69.5.2 Convergence Rate

How long does it take for time series sample averages to converge to cross-sectional averages?

We can use QuantEcon.py's MarkovChain class to investigate this.

Let's plot the path of the sample averages over 5,000 periods

In [10]:
```
lm = LakeModel(d=0, b=0)
T = 5000  # Simulation length

α, λ = lm.α, lm.λ

P = [[1 - λ,         λ],
     [    α,     1 - α]]

mc = MarkovChain(P)

xbar = lm.rate_steady_state()

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
s_path = mc.simulate(T, init=1)
s_bar_e = s_path.cumsum() / range(1, T+1)
s_bar_u = 1 - s_bar_e

to_plot = [s_bar_u, s_bar_e]
titles = ['Percent of time unemployed', 'Percent of time employed']

for i, plot in enumerate(to_plot):
    axes[i].plot(plot, lw=2, alpha=0.5)
    axes[i].hlines(xbar[i], 0, T, 'r', '--')
    axes[i].set_title(titles[i])
    axes[i].grid()

plt.tight_layout()
plt.show()
```

The stationary probabilities are given by the dashed red line.

In this case it takes much of the sample for these two objects to converge.

This is largely due to the high persistence in the Markov chain.

69.6 Endogenous Job Finding Rate

We now make the hiring rate endogenous.

The transition rate from unemployment to employment will be determined by the McCall search model [116].

All details relevant to the following discussion can be found in our treatment of that model.

69.6.1 Reservation Wage

The most important thing to remember about the model is that optimal decisions are characterized by a reservation wage $\bar{w}$

- If the wage offer w in hand is greater than or equal to $\bar{w}$, then the worker accepts.
- Otherwise, the worker rejects.

As we saw in our discussion of the model, the reservation wage depends on the wage offer distribution and the parameters

- α, the separation rate

- β, the discount factor
- γ, the offer arrival rate
- c, unemployment compensation

69.6.2 Linking the McCall Search Model to the Lake Model

Suppose that all workers inside a lake model behave according to the McCall search model.

The exogenous probability of leaving employment remains α.

But their optimal decision rules determine the probability λ of leaving unemployment.

This is now

$$\lambda = \gamma \mathbb{P}\{w_t \geq \bar{w}\} = \gamma \sum_{w' \geq \bar{w}} p(w') \tag{1}$$

69.6.3 Fiscal Policy

We can use the McCall search version of the Lake Model to find an optimal level of unemployment insurance.

We assume that the government sets unemployment compensation c.

The government imposes a lump-sum tax τ sufficient to finance total unemployment payments.

To attain a balanced budget at a steady state, taxes, the steady state unemployment rate u, and the unemployment compensation rate must satisfy

$$\tau = uc$$

The lump-sum tax applies to everyone, including unemployed workers.

Thus, the post-tax income of an employed worker with wage w is $w - \tau$.

The post-tax income of an unemployed worker is $c - \tau$.

For each specification (c, τ) of government policy, we can solve for the worker's optimal reservation wage.

This determines λ via (1) evaluated at post tax wages, which in turn determines a steady state unemployment rate $u(c, \tau)$.

For a given level of unemployment benefit c, we can solve for a tax that balances the budget in the steady state

$$\tau = u(c, \tau)c$$

To evaluate alternative government tax-unemployment compensation pairs, we require a welfare criterion.

We use a steady state welfare criterion

$$W := e\, \mathbb{E}[V \,|\, \text{employed}] + u\, U$$

where the notation V and U is as defined in the McCall search model lecture.

The wage offer distribution will be a discretized version of the lognormal distribution $LN(\log(20), 1)$, as shown in the next figure

We take a period to be a month.

We set b and d to match monthly birth and death rates, respectively, in the U.S. population

- $b = 0.0124$
- $d = 0.00822$

Following [39], we set α, the hazard rate of leaving employment, to

- $\alpha = 0.013$

69.6.4 Fiscal Policy Code

We will make use of techniques from the McCall model lecture

The first piece of code implements value function iteration

In [11]:
```
# A default utility function

@jit
def u(c, σ):
    if c > 0:
        return (c**(1 - σ) - 1) / (1 - σ)
    else:
        return -10e6


class McCallModel:
    """
    Stores the parameters and functions associated with a given model.
    """

    def __init__(self,
                 α=0.2,        # Job separation rate
```

```
                β=0.98,       # Discount rate
                γ=0.7,        # Job offer rate
                c=6.0,        # Unemployment compensation
                σ=2.0,        # Utility parameter
                w_vec=None,   # Possible wage values
                p_vec=None): # Probabilities over w_vec

        self.α, self.β, self.γ, self.c = α, β, γ, c
        self.σ = σ

       # Add a default wage vector and probabilities over the vector using
        # the beta-binomial distribution
        if w_vec is None:
            n = 60  # Number of possible outcomes for wage
            # Wages between 10 and 20
            self.w_vec = np.linspace(10, 20, n)
            a, b = 600, 400  # Shape parameters
            dist = BetaBinomial(n-1, a, b)
            self.p_vec = dist.pdf()
        else:
            self.w_vec = w_vec
            self.p_vec = p_vec

@jit
def _update_bellman(α, β, γ, c, σ, w_vec, p_vec, V, V_new, U):
    """
    A jitted function to update the Bellman equations.  Note that V_new is
    modified in place (i.e, modified by this function).  The new value of U
    is returned.

    """
    for w_idx, w in enumerate(w_vec):
        # w_idx indexes the vector of possible wages
        V_new[w_idx] = u(w, σ) + β * ((1 - α) * V[w_idx] + α * U)

    U_new = u(c, σ) + β * (1 - γ) * U + \
                    β * γ * np.sum(np.maximum(U, V) * p_vec)

    return U_new


def solve_mccall_model(mcm, tol=1e-5, max_iter=2000):
    """
    Iterates to convergence on the Bellman equations

    Parameters
    ----------
    mcm : an instance of McCallModel
    tol : float
        error tolerance
    max_iter : int
        the maximum number of iterations
    """

    V = np.ones(len(mcm.w_vec))  # Initial guess of V
    V_new = np.empty_like(V)     # To store updates to V
    U = 1                        # Initial guess of U
    i = 0
```

```
    error = tol + 1

    while error > tol and i < max_iter:
        U_new = _update_bellman(mcm.α, mcm.β, mcm.γ,
                mcm.c, mcm.σ, mcm.w_vec, mcm.p_vec, V, V_new, U)
        error_1 = np.max(np.abs(V_new - V))
        error_2 = np.abs(U_new - U)
        error = max(error_1, error_2)
        V[:] = V_new
        U = U_new
        i += 1

    return V, U
```

The second piece of code is used to complete the reservation wage:

```
In [12]: def compute_reservation_wage(mcm, return_values=False):
             """
             Computes the reservation wage of an instance of the McCall model
             by finding the smallest w such that V(w) > U.

             If V(w) > U for all w, then the reservation wage w_bar is set to
             the lowest wage in mcm.w_vec.

             If v(w) < U for all w, then w_bar is set to np.inf.

             Parameters
             ----------
             mcm : an instance of McCallModel
             return_values : bool (optional, default=False)
                 Return the value functions as well

             Returns
             -------
             w_bar : scalar
                 The reservation wage

             """

             V, U = solve_mccall_model(mcm)
             w_idx = np.searchsorted(V - U, 0)

             if w_idx == len(V):
                 w_bar = np.inf
             else:
                 w_bar = mcm.w_vec[w_idx]

             if return_values == False:
                 return w_bar
             else:
                 return w_bar, V, U
```

Now let's compute and plot welfare, employment, unemployment, and tax revenue as a function of the unemployment compensation rate

```
In [13]: # Some global variables that will stay constant
         α = 0.013
```

```
α_q = (1-(1-α)**3)   # Quarterly (α is monthly)
b = 0.0124
d = 0.00822
β = 0.98
γ = 1.0
σ = 2.0

# The default wage distribution --- a discretized lognormal
log_wage_mean, wage_grid_size, max_wage = 20, 200, 170
logw_dist = norm(np.log(log_wage_mean), 1)
w_vec = np.linspace(1e-8, max_wage, wage_grid_size + 1)
cdf = logw_dist.cdf(np.log(w_vec))
pdf = cdf[1:] - cdf[:-1]
p_vec = pdf / pdf.sum()
w_vec = (w_vec[1:] + w_vec[:-1]) / 2


def compute_optimal_quantities(c, τ):
    """
    Compute the reservation wage, job finding rate and value functions
    of the workers given c and τ.

    """

    mcm = McCallModel(α=α_q,
                      β=β,
                      γ=γ,
                      c=c-τ,           # Post tax compensation
                      σ=σ,
                      w_vec=w_vec-τ,   # Post tax wages
                      p_vec=p_vec)

    w_bar, V, U = compute_reservation_wage(mcm, return_values=True)
    λ = γ * np.sum(p_vec[w_vec - τ > w_bar])
    return w_bar, λ, V, U

def compute_steady_state_quantities(c, τ):
    """
    Compute the steady state unemployment rate given c and τ using optimal
    quantities from the McCall model and computing corresponding steady
    state quantities

    """
    w_bar, λ, V, U = compute_optimal_quantities(c, τ)

    # Compute steady state employment and unemployment rates
    lm = LakeModel(α=α_q, λ=λ, b=b, d=d)
    x = lm.rate_steady_state()
    u, e = x

    # Compute steady state welfare
    w = np.sum(V * p_vec * (w_vec - τ > w_bar)) / np.sum(p_vec * (w_vec -
            τ > w_bar))
    welfare = e * w + u * U

    return e, u, welfare
```

```python
def find_balanced_budget_tax(c):
    """
    Find the tax level that will induce a balanced budget.

    """
    def steady_state_budget(t):
        e, u, w = compute_steady_state_quantities(c, t)
        return t - u * c

    τ = brentq(steady_state_budget, 0.0, 0.9 * c)
    return τ


# Levels of unemployment insurance we wish to study
c_vec = np.linspace(5, 140, 60)

tax_vec = []
unempl_vec = []
empl_vec = []
welfare_vec = []

for c in c_vec:
    t = find_balanced_budget_tax(c)
    e_rate, u_rate, welfare = compute_steady_state_quantities(c, t)
    tax_vec.append(t)
    unempl_vec.append(u_rate)
    empl_vec.append(e_rate)
    welfare_vec.append(welfare)

fig, axes = plt.subplots(2, 2, figsize=(12, 10))

plots = [unempl_vec, empl_vec, tax_vec, welfare_vec]
titles = ['Unemployment', 'Employment', 'Tax', 'Welfare']

for ax, plot, title in zip(axes.flatten(), plots, titles):
    ax.plot(c_vec, plot, lw=2, alpha=0.7)
    ax.set_title(title)
    ax.grid()

plt.tight_layout()
plt.show()
```

Welfare first increases and then decreases as unemployment benefits rise.

The level that maximizes steady state welfare is approximately 62.

69.7 Exercises

69.7.1 Exercise 1

Consider an economy with an initial stock of workers $N_0 = 100$ at the steady state level of employment in the baseline parameterization

- $\alpha = 0.013$
- $\lambda = 0.283$
- $b = 0.0124$
- $d = 0.00822$

(The values for α and λ follow [39])

Suppose that in response to new legislation the hiring rate reduces to $\lambda = 0.2$.

Plot the transition dynamics of the unemployment and employment stocks for 50 periods.

Plot the transition dynamics for the rates.

How long does the economy take to converge to its new steady state?

What is the new steady state level of employment?

69.7.2 Exercise 2

Consider an economy with an initial stock of workers $N_0 = 100$ at the steady state level of employment in the baseline parameterization.

Suppose that for 20 periods the birth rate was temporarily high ($b = 0.0025$) and then returned to its original level.

Plot the transition dynamics of the unemployment and employment stocks for 50 periods.

Plot the transition dynamics for the rates.

How long does the economy take to return to its original steady state?

69.8 Solutions

69.9 Lake Model Solutions

69.9.1 Exercise 1

We begin by constructing the class containing the default parameters and assigning the steady state values to x0

```
In [14]: lm = LakeModel()
         x0 = lm.rate_steady_state()
         print(f"Initial Steady State: {x0}")
```

```
Initial Steady State: [0.08266806 0.91733194]
```

Initialize the simulation values

```
In [15]: N0 = 100
         T = 50
```

New legislation changes λ to 0.2

```
In [16]: lm.lmda = 0.2

         xbar = lm.rate_steady_state()  # new steady state
         X_path = np.vstack(tuple(lm.simulate_stock_path(x0 * N0, T)))
         x_path = np.vstack(tuple(lm.simulate_rate_path(x0, T)))
         print(f"New Steady State: {xbar}")
```

```
New Steady State: [0.08266806 0.91733194]
```

Now plot stocks

```
In [17]: fig, axes = plt.subplots(3, 1, figsize=[10, 9])
```

```
axes[0].plot(X_path[:, 0])
axes[0].set_title('Unemployment')

axes[1].plot(X_path[:, 1])
axes[1].set_title('Employment')

axes[2].plot(X_path.sum(1))
axes[2].set_title('Labor force')

for ax in axes:
    ax.grid()

plt.tight_layout()
plt.show()
```

And how the rates evolve

```
In [18]: fig, axes = plt.subplots(2, 1, figsize=(10, 8))

         titles = ['Unemployment rate', 'Employment rate']

         for i, title in enumerate(titles):
             axes[i].plot(x_path[:, i])
             axes[i].hlines(xbar[i], 0, T, 'r', '--')
```

```
        axes[i].set_title(title)
        axes[i].grid()

    plt.tight_layout()
    plt.show()
```

We see that it takes 20 periods for the economy to converge to its new steady state levels.

69.9.2 Exercise 2

This next exercise has the economy experiencing a boom in entrances to the labor market and then later returning to the original levels.

For 20 periods the economy has a new entry rate into the labor market.

Let's start off at the baseline parameterization and record the steady state

```
In [19]: lm = LakeModel()
         x0 = lm.rate_steady_state()
```

Here are the other parameters:

```
In [20]: b_hat = 0.003
         T_hat = 20
```

Let's increase b to the new value and simulate for 20 periods

```
In [21]: lm.b = b_hat
         # Simulate stocks
         X_path1 = np.vstack(tuple(lm.simulate_stock_path(x0 * N0, T_hat)))
         # Simulate rates
         x_path1 = np.vstack(tuple(lm.simulate_rate_path(x0, T_hat)))
```

Now we reset b to the original value and then, using the state after 20 periods for the new initial conditions, we simulate for the additional 30 periods

```
In [22]: lm.b = 0.0124
         # Simulate stocks
         X_path2 = np.vstack(tuple(lm.simulate_stock_path(X_path1[-1, :2], T-
T_hat+1)))
         # Simulate rates
         x_path2 = np.vstack(tuple(lm.simulate_rate_path(x_path1[-1, :2], T-
T_hat+1)))
```

Finally, we combine these two paths and plot

```
In [23]: # note [1:] to avoid doubling period 20
         x_path = np.vstack([x_path1, x_path2[1:]])
         X_path = np.vstack([X_path1, X_path2[1:]])

         fig, axes = plt.subplots(3, 1, figsize=[10, 9])

         axes[0].plot(X_path[:, 0])
         axes[0].set_title('Unemployment')

         axes[1].plot(X_path[:, 1])
         axes[1].set_title('Employment')

         axes[2].plot(X_path.sum(1))
         axes[2].set_title('Labor force')

         for ax in axes:
             ax.grid()

         plt.tight_layout()
         plt.show()
```

And the rates

```
In [24]: fig, axes = plt.subplots(2, 1, figsize=[10, 6])

         titles = ['Unemployment rate', 'Employment rate']

         for i, title in enumerate(titles):
             axes[i].plot(x_path[:, i])
             axes[i].hlines(x0[i], 0, T, 'r', '--')
             axes[i].set_title(title)
             axes[i].grid()

         plt.tight_layout()
         plt.show()
```

Unemployment rate
0.080
0.075
0.070
0.065
0.060
0.055
0
10
20
30
40
50
Employment rate
0.945
0.940
0.935
0.930
0.925
0.920
0
10
20
30
40
50

Chapter 70

Rational Expectations Equilibrium

70.1 Contents

- Overview 70.2
- Defining Rational Expectations Equilibrium 70.3
- Computation of an Equilibrium 70.4
- Exercises 70.5
- Solutions 70.6

> "If you're so smart, why aren't you rich?"

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

70.2 Overview

This lecture introduces the concept of *rational expectations equilibrium*.

To illustrate it, we describe a linear quadratic version of a famous and important model due to Lucas and Prescott [110].

This 1971 paper is one of a small number of research articles that kicked off the *rational expectations revolution*.

We follow Lucas and Prescott by employing a setting that is readily "Bellmanized" (i.e., capable of being formulated in terms of dynamic programming problems).

Because we use linear quadratic setups for demand and costs, we can adapt the LQ programming techniques described in this lecture.

We will learn about how a representative agent's problem differs from a planner's, and how a planning problem can be used to compute rational expectations quantities.

We will also learn about how a rational expectations equilibrium can be characterized as a fixed point of a mapping from a *perceived law of motion* to an *actual law of motion*.

Equality between a perceived and an actual law of motion for endogenous market-wide objects captures in a nutshell what the rational expectations equilibrium concept is all about.

Finally, we will learn about the important "Big K, little k" trick, a modeling device widely used in macroeconomics.

Except that for us

- Instead of "Big K" it will be "Big Y".
- Instead of "little k" it will be "little y".

Let's start with some standard imports:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

We'll also use the LQ class from QuantEcon.py.

```
In [3]: from quantecon import LQ
```

70.2.1 The Big Y, Little y Trick

This widely used method applies in contexts in which a "representative firm" or agent is a "price taker" operating within a competitive equilibrium.

We want to impose that

- The representative firm or individual takes *aggregate* Y as given when it chooses individual y, but ....
- At the end of the day, $Y = y$, so that the representative firm is indeed representative.

The Big Y, little y trick accomplishes these two goals by

- Taking Y as beyond control when posing the choice problem of who chooses y; but ....
- Imposing $Y = y$ *after* having solved the individual's optimization problem.

Please watch for how this strategy is applied as the lecture unfolds.

We begin by applying the Big Y, little y trick in a very simple static context.

A Simple Static Example of the Big Y, Little y Trick

Consider a static model in which a collection of n firms produce a homogeneous good that is sold in a competitive market.

Each of these n firms sell output y.

The price p of the good lies on an inverse demand curve

$$p = a_0 - a_1 Y \tag{1}$$

where

- $a_i > 0$ for $i = 0, 1$
- $Y = ny$ is the market-wide level of output

Each firm has a total cost function

$$c(y) = c_1 y + 0.5 c_2 y^2, \qquad c_i > 0 \text{ for } i = 1, 2$$

The profits of a representative firm are $py - c(y)$.

Using (1), we can express the problem of the representative firm as

$$\max_y \Big[(a_0 - a_1 Y) y - c_1 y - 0.5 c_2 y^2 \Big] \tag{2}$$

In posing problem (2), we want the firm to be a *price taker*.

We do that by regarding p and therefore Y as exogenous to the firm.

The essence of the Big Y, little y trick is *not* to set $Y = ny$ *before* taking the first-order condition with respect to y in problem (2).

This assures that the firm is a price taker.

The first-order condition for problem (2) is

$$a_0 - a_1 Y - c_1 - c_2 y = 0 \tag{3}$$

At this point, *but not before*, we substitute $Y = ny$ into (3) to obtain the following linear equation

$$a_0 - c_1 - (a_1 + n^{-1} c_2) Y = 0 \tag{4}$$

to be solved for the competitive equilibrium market-wide output Y.

After solving for Y, we can compute the competitive equilibrium price p from the inverse demand curve (1).

70.2.2 Further Reading

References for this lecture include

- [110]
- [142], chapter XIV
- [108], chapter 7

70.3 Defining Rational Expectations Equilibrium

Our first illustration of a rational expectations equilibrium involves a market with n firms, each of which seeks to maximize the discounted present value of profits in the face of adjustment costs.

The adjustment costs induce the firms to make gradual adjustments, which in turn requires consideration of future prices.

Individual firms understand that, via the inverse demand curve, the price is determined by the amounts supplied by other firms.

Hence each firm wants to forecast future total industry supplies.

In our context, a forecast is generated by a belief about the law of motion for the aggregate state.

Rational expectations equilibrium prevails when this belief coincides with the actual law of motion generated by production choices induced by this belief.

We formulate a rational expectations equilibrium in terms of a fixed point of an operator that maps beliefs into optimal beliefs.

70.3.1 Competitive Equilibrium with Adjustment Costs

To illustrate, consider a collection of n firms producing a homogeneous good that is sold in a competitive market.

Each of these n firms sell output y_t.

The price p_t of the good lies on the inverse demand curve

$$p_t = a_0 - a_1 Y_t \tag{5}$$

where

- $a_i > 0$ for $i = 0, 1$
- $Y_t = n y_t$ is the market-wide level of output

The Firm's Problem

Each firm is a price taker.

While it faces no uncertainty, it does face adjustment costs

In particular, it chooses a production plan to maximize

$$\sum_{t=0}^{\infty} \beta^t r_t \tag{6}$$

where

$$r_t := p_t y_t - \frac{\gamma (y_{t+1} - y_t)^2}{2}, \qquad y_0 \text{ given} \tag{7}$$

Regarding the parameters,

- $\beta \in (0, 1)$ is a discount factor
- $\gamma > 0$ measures the cost of adjusting the rate of output

Regarding timing, the firm observes p_t and y_t when it chooses y_{t+1} at time t.

To state the firm's optimization problem completely requires that we specify dynamics for all state variables.

This includes ones that the firm cares about but does not control like p_t.

We turn to this problem now.

Prices and Aggregate Output

In view of (5), the firm's incentive to forecast the market price translates into an incentive to forecast aggregate output Y_t.

Aggregate output depends on the choices of other firms.

We assume that n is such a large number that the output of any single firm has a negligible effect on aggregate output.

That justifies firms in regarding their forecasts of aggregate output as being unaffected by their own output decisions.

The Firm's Beliefs

We suppose the firm believes that market-wide output Y_t follows the law of motion

$$Y_{t+1} = H(Y_t) \tag{8}$$

where Y_0 is a known initial condition.

The *belief function* H is an equilibrium object, and hence remains to be determined.

Optimal Behavior Given Beliefs

For now, let's fix a particular belief H in (8) and investigate the firm's response to it.

Let v be the optimal value function for the firm's problem given H.

The value function satisfies the Bellman equation

$$v(y, Y) = \max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \tag{9}$$

Let's denote the firm's optimal policy function by h, so that

$$y_{t+1} = h(y_t, Y_t) \tag{10}$$

where

$$h(y, Y) := \operatorname*{argmax}_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \tag{11}$$

Evidently v and h both depend on H.

A First-Order Characterization

In what follows it will be helpful to have a second characterization of h, based on first-order conditions.

The first-order necessary condition for choosing y' is

$$-\gamma(y' - y) + \beta v_y(y', H(Y)) = 0 \tag{12}$$

An important useful envelope result of Benveniste-Scheinkman [19] implies that to differentiate v with respect to y we can naively differentiate the right side of (9), giving

$$v_y(y, Y) = a_0 - a_1 Y + \gamma(y' - y)$$

Substituting this equation into (12) gives the *Euler equation*

$$-\gamma(y_{t+1} - y_t) + \beta[a_0 - a_1 Y_{t+1} + \gamma(y_{t+2} - y_{t+1})] = 0 \tag{13}$$

The firm optimally sets an output path that satisfies (13), taking (8) as given, and subject to

- the initial conditions for (y_0, Y_0).
- the terminal condition $\lim_{t\to\infty} \beta^t y_t v_y(y_t, Y_t) = 0$.

This last condition is called the *transversality condition*, and acts as a first-order necessary condition "at infinity".

The firm's decision rule solves the difference equation (13) subject to the given initial condition y_0 and the transversality condition.

Note that solving the Bellman equation (9) for v and then h in (11) yields a decision rule that automatically imposes both the Euler equation (13) and the transversality condition.

The Actual Law of Motion for Output

As we've seen, a given belief translates into a particular decision rule h.

Recalling that $Y_t = ny_t$, the *actual law of motion* for market-wide output is then

$$Y_{t+1} = nh(Y_t/n, Y_t) \tag{14}$$

Thus, when firms believe that the law of motion for market-wide output is (8), their optimizing behavior makes the actual law of motion be (14).

70.3.2 Definition of Rational Expectations Equilibrium

A *rational expectations equilibrium* or *recursive competitive equilibrium* of the model with adjustment costs is a decision rule h and an aggregate law of motion H such that

1. Given belief H, the map h is the firm's optimal policy function.

2. The law of motion H satisfies $H(Y) = nh(Y/n, Y)$ for all Y.

Thus, a rational expectations equilibrium equates the perceived and actual laws of motion (8) and (14).

Fixed Point Characterization

As we've seen, the firm's optimum problem induces a mapping Φ from a perceived law of motion H for market-wide output to an actual law of motion $\Phi(H)$.

The mapping Φ is the composition of two operations, taking a perceived law of motion into a decision rule via (9)–(11), and a decision rule into an actual law via (14).

The H component of a rational expectations equilibrium is a fixed point of Φ.

70.4 Computation of an Equilibrium

Now let's consider the problem of computing the rational expectations equilibrium.

70.4.1 Failure of Contractivity

Readers accustomed to dynamic programming arguments might try to address this problem by choosing some guess H_0 for the aggregate law of motion and then iterating with Φ.

Unfortunately, the mapping Φ is not a contraction.

In particular, there is no guarantee that direct iterations on Φ converge Section **??**.

Furthermore, there are examples in which these iterations diverge.

Fortunately, there is another method that works here.

The method exploits a connection between equilibrium and Pareto optimality expressed in the fundamental theorems of welfare economics (see, e.g, [115]).

Lucas and Prescott [110] used this method to construct a rational expectations equilibrium.

The details follow.

70.4.2 A Planning Problem Approach

Our plan of attack is to match the Euler equations of the market problem with those for a single-agent choice problem.

As we'll see, this planning problem can be solved by LQ control (linear regulator).

The optimal quantities from the planning problem are rational expectations equilibrium quantities.

The rational expectations equilibrium price can be obtained as a shadow price in the planning problem.

For convenience, in this section, we set $n = 1$.

We first compute a sum of consumer and producer surplus at time t

$$s(Y_t, Y_{t+1}) := \int_0^{Y_t} (a_0 - a_1 x)\, dx - \frac{\gamma(Y_{t+1} - Y_t)^2}{2} \tag{15}$$

The first term is the area under the demand curve, while the second measures the social costs of changing output.

The *planning problem* is to choose a production plan $\{Y_t\}$ to maximize

$$\sum_{t=0}^{\infty} \beta^t s(Y_t, Y_{t+1})$$

subject to an initial condition for Y_0.

70.4.3 Solution of the Planning Problem

Evaluating the integral in (15) yields the quadratic form $a_0 Y_t - a_1 Y_t^2/2$.

As a result, the Bellman equation for the planning problem is

$$V(Y) = \max_{Y'} \left\{ a_0 Y - \frac{a_1}{2} Y^2 - \frac{\gamma(Y' - Y)^2}{2} + \beta V(Y') \right\} \tag{16}$$

The associated first-order condition is

$$-\gamma(Y' - Y) + \beta V'(Y') = 0 \tag{17}$$

Applying the same Benveniste-Scheinkman formula gives

$$V'(Y) = a_0 - a_1 Y + \gamma(Y' - Y)$$

Substituting this into equation (17) and rearranging leads to the Euler equation

$$\beta a_0 + \gamma Y_t - [\beta a_1 + \gamma(1 + \beta)] Y_{t+1} + \gamma \beta Y_{t+2} = 0 \tag{18}$$

70.4.4 The Key Insight

Return to equation (13) and set $y_t = Y_t$ for all t.

(Recall that for this section we've set $n = 1$ to simplify the calculations)

A small amount of algebra will convince you that when $y_t = Y_t$, equations (18) and (13) are identical.

Thus, the Euler equation for the planning problem matches the second-order difference equation that we derived by

1. finding the Euler equation of the representative firm and
2. substituting into it the expression $Y_t = n y_t$ that "makes the representative firm be representative".

If it is appropriate to apply the same terminal conditions for these two difference equations, which it is, then we have verified that a solution of the planning problem is also a rational expectations equilibrium quantity sequence.

It follows that for this example we can compute equilibrium quantities by forming the optimal linear regulator problem corresponding to the Bellman equation (16).

The optimal policy function for the planning problem is the aggregate law of motion H that the representative firm faces within a rational expectations equilibrium.

Structure of the Law of Motion

As you are asked to show in the exercises, the fact that the planner's problem is an LQ problem implies an optimal policy — and hence aggregate law of motion — taking the form

$$Y_{t+1} = \kappa_0 + \kappa_1 Y_t \tag{19}$$

for some parameter pair κ_0, κ_1.

Now that we know the aggregate law of motion is linear, we can see from the firm's Bellman equation (9) that the firm's problem can also be framed as an LQ problem.

As you're asked to show in the exercises, the LQ formulation of the firm's problem implies a law of motion that looks as follows

$$y_{t+1} = h_0 + h_1 y_t + h_2 Y_t \tag{20}$$

Hence a rational expectations equilibrium will be defined by the parameters $(\kappa_0, \kappa_1, h_0, h_1, h_2)$ in (19)–(20).

70.5 Exercises

70.5.1 Exercise 1

Consider the firm problem described above.

Let the firm's belief function H be as given in (19).

Formulate the firm's problem as a discounted optimal linear regulator problem, being careful to describe all of the objects needed.

Use the class `LQ` from the QuantEcon.py package to solve the firm's problem for the following parameter values:

$$a_0 = 100, a_1 = 0.05, \beta = 0.95, \gamma = 10, \kappa_0 = 95.5, \kappa_1 = 0.95$$

Express the solution of the firm's problem in the form (20) and give the values for each h_j.

If there were n identical competitive firms all behaving according to (20), what would (20) imply for the *actual* law of motion (8) for market supply.

70.5.2 Exercise 2

Consider the following κ_0, κ_1 pairs as candidates for the aggregate law of motion component of a rational expectations equilibrium (see (19)).

Extending the program that you wrote for exercise 1, determine which if any satisfy the definition of a rational expectations equilibrium

- (94.0886298678, 0.923409232937)
- (93.2119845412, 0.984323478873)
- (95.0818452486, 0.952459076301)

Describe an iterative algorithm that uses the program that you wrote for exercise 1 to compute a rational expectations equilibrium.

(You are not being asked actually to use the algorithm you are suggesting)

70.5.3 Exercise 3

Recall the planner's problem described above

1. Formulate the planner's problem as an LQ problem.
2. Solve it using the same parameter values in exercise 1

- $a_0 = 100, a_1 = 0.05, \beta = 0.95, \gamma = 10$

1. Represent the solution in the form $Y_{t+1} = \kappa_0 + \kappa_1 Y_t$.
2. Compare your answer with the results from exercise 2.

70.5.4 Exercise 4

A monopolist faces the industry demand curve (5) and chooses $\{Y_t\}$ to maximize $\sum_{t=0}^{\infty} \beta^t r_t$ where

$$r_t = p_t Y_t - \frac{\gamma(Y_{t+1} - Y_t)^2}{2}$$

Formulate this problem as an LQ problem.

Compute the optimal policy using the same parameters as the previous exercise.

In particular, solve for the parameters in

$$Y_{t+1} = m_0 + m_1 Y_t$$

Compare your results with the previous exercise – comment.

70.6 Solutions

70.6.1 Exercise 1

To map a problem into a discounted optimal linear control problem, we need to define

- state vector x_t and control vector u_t
- matrices A, B, Q, R that define preferences and the law of motion for the state

For the state and control vectors, we choose

$$x_t = \begin{bmatrix} y_t \\ Y_t \\ 1 \end{bmatrix}, \qquad u_t = y_{t+1} - y_t$$

For B, Q, R we set

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \kappa_1 & \kappa_0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad R = \begin{bmatrix} 0 & a_1/2 & -a_0/2 \\ a_1/2 & 0 & 0 \\ -a_0/2 & 0 & 0 \end{bmatrix}, \quad Q = \gamma/2$$

By multiplying out you can confirm that

- $x_t' R x_t + u_t' Q u_t = -r_t$
- $x_{t+1} = A x_t + B u_t$

We'll use the module `lqcontrol.py` to solve the firm's problem at the stated parameter values.

This will return an LQ policy F with the interpretation $u_t = -F x_t$, or

$$y_{t+1} - y_t = -F_0 y_t - F_1 Y_t - F_2$$

Matching parameters with $y_{t+1} = h_0 + h_1 y_t + h_2 Y_t$ leads to

$$h_0 = -F_2, \quad h_1 = 1 - F_0, \quad h_2 = -F_1$$

Here's our solution

In [4]:
```
# Model parameters

a0 = 100
a1 = 0.05
β = 0.95
γ = 10.0

# Beliefs

κ0 = 95.5
κ1 = 0.95

# Formulate the LQ problem

A = np.array([[1, 0, 0], [0, κ1, κ0], [0, 0, 1]])
B = np.array([1, 0, 0])
B.shape = 3, 1
R = np.array([[0, a1/2, -a0/2], [a1/2, 0, 0], [-a0/2, 0, 0]])
Q = 0.5 * γ

# Solve for the optimal policy

lq = LQ(Q, R, A, B, beta=β)
P, F, d = lq.stationary_values()
F = F.flatten()
```

```
out1 = f"F = [{F[0]:.3f}, {F[1]:.3f}, {F[2]:.3f}]"
h0, h1, h2 = -F[2], 1 - F[0], -F[1]
out2 = f"(h0, h1, h2) = ({h0:.3f}, {h1:.3f}, {h2:.3f})"

print(out1)
print(out2)
```

```
   F = [-0.000, 0.046, -96.949]
(h0, h1, h2) = (96.949, 1.000, -0.046)
```

The implication is that

$$y_{t+1} = 96.949 + y_t - 0.046\,Y_t$$

For the case $n > 1$, recall that $Y_t = ny_t$, which, combined with the previous equation, yields

$$Y_{t+1} = n\left(96.949 + y_t - 0.046\,Y_t\right) = n96.949 + (1 - n0.046)Y_t$$

70.6.2 Exercise 2

To determine whether a κ_0, κ_1 pair forms the aggregate law of motion component of a rational expectations equilibrium, we can proceed as follows:

- Determine the corresponding firm law of motion $y_{t+1} = h_0 + h_1 y_t + h_2 Y_t$.
- Test whether the associated aggregate law :$Y_{t+1} = nh(Y_t/n, Y_t)$ evaluates to $Y_{t+1} = \kappa_0 + \kappa_1 Y_t$.

In the second step, we can use $Y_t = ny_t = y_t$, so that $Y_{t+1} = nh(Y_t/n, Y_t)$ becomes

$$Y_{t+1} = h(Y_t, Y_t) = h_0 + (h_1 + h_2)Y_t$$

Hence to test the second step we can test $\kappa_0 = h_0$ and $\kappa_1 = h_1 + h_2$.

The following code implements this test

```
In [5]: candidates = ((94.0886298678, 0.923409232937),
                      (93.2119845412, 0.984323478873),
                      (95.0818452486, 0.952459076301))

        for ϰ0, ϰ1 in candidates:

            # Form the associated law of motion
            A = np.array([[1, 0, 0], [0, ϰ1, ϰ0], [0, 0, 1]])

            # Solve the LQ problem for the firm
            lq = LQ(Q, R, A, B, beta=β)
            P, F, d = lq.stationary_values()
            F = F.flatten()
            h0, h1, h2 = -F[2], 1 - F[0], -F[1]

            # Test the equilibrium condition
            if np.allclose((ϰ0, ϰ1), (h0, h1 + h2)):
```

```
        print(f'Equilibrium pair = {x0}, {x1}')
        print('f(h0, h1, h2) = {h0}, {h1}, {h2}')
        break
```

```
    Equilibrium pair = 95.0818452486, 0.952459076301
f(h0, h1, h2) = {h0}, {h1}, {h2}
```

The output tells us that the answer is pair (iii), which implies $(h_0, h_1, h_2) = (95.0819, 1.0000, -.0475)$.

(Notice we use `np.allclose` to test equality of floating-point numbers, since exact equality is too strict).

Regarding the iterative algorithm, one could loop from a given (κ_0, κ_1) pair to the associated firm law and then to a new (κ_0, κ_1) pair.

This amounts to implementing the operator Φ described in the lecture.

(There is in general no guarantee that this iterative process will converge to a rational expectations equilibrium)

70.6.3 Exercise 3

We are asked to write the planner problem as an LQ problem.

For the state and control vectors, we choose

$$x_t = \begin{bmatrix} Y_t \\ 1 \end{bmatrix}, \quad u_t = Y_{t+1} - Y_t$$

For the LQ matrices, we set

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad R = \begin{bmatrix} a_1/2 & -a_0/2 \\ -a_0/2 & 0 \end{bmatrix}, \quad Q = \gamma/2$$

By multiplying out you can confirm that

- $x_t' R x_t + u_t' Q u_t = -s(Y_t, Y_{t+1})$
- $x_{t+1} = A x_t + B u_t$

By obtaining the optimal policy and using $u_t = -F x_t$ or

$$Y_{t+1} - Y_t = -F_0 Y_t - F_1$$

we can obtain the implied aggregate law of motion via $\kappa_0 = -F_1$ and $\kappa_1 = 1 - F_0$.

The Python code to solve this problem is below:

```
In [6]: # Formulate the planner's LQ problem

        A = np.array([[1, 0], [0, 1]])
        B = np.array([[1], [0]])
        R = np.array([[a1 / 2, -a0 / 2], [-a0 / 2, 0]])
        Q = γ / 2
```

```
# Solve for the optimal policy

lq = LQ(Q, R, A, B, beta=β)
P, F, d = lq.stationary_values()

# Print the results

F = F.flatten()
ϰ0, ϰ1 = -F[1], 1 - F[0]
print(ϰ0, ϰ1)
```

```
95.08187459215002 0.9524590627039248
```

The output yields the same (κ_0, κ_1) pair obtained as an equilibrium from the previous exercise.

70.6.4 Exercise 4

The monopolist's LQ problem is almost identical to the planner's problem from the previous exercise, except that

$$R = \begin{bmatrix} a_1 & -a_0/2 \\ -a_0/2 & 0 \end{bmatrix}$$

The problem can be solved as follows

```
In [7]: A = np.array([[1, 0], [0, 1]])
        B = np.array([[1], [0]])
        R = np.array([[a1, -a0 / 2], [-a0 / 2, 0]])
        Q = γ / 2

        lq = LQ(Q, R, A, B, beta=β)
        P, F, d = lq.stationary_values()

        F = F.flatten()
        m0, m1 = -F[1], 1 - F[0]
        print(m0, m1)
```

```
73.47294403502818 0.9265270559649701
```

We see that the law of motion for the monopolist is approximately $Y_{t+1} = 73.4729 + 0.9265 Y_t$.

In the rational expectations case, the law of motion was approximately $Y_{t+1} = 95.0818 + 0.9525 Y_t$.

One way to compare these two laws of motion is by their fixed points, which give long-run equilibrium output in each case.

For laws of the form $Y_{t+1} = c_0 + c_1 Y_t$, the fixed point is $c_0/(1 - c_1)$.

If you crunch the numbers, you will see that the monopolist adopts a lower long-run quantity than obtained by the competitive market, implying a higher market price.

This is analogous to the elementary static-case results

Footnotes

[1] A literature that studies whether models populated with agents who learn can converge to rational expectations equilibria features iterations on a modification of the mapping Φ that can be approximated as $\gamma\Phi + (1-\gamma)I$. Here I is the identity operator and $\gamma \in (0,1)$ is a *relaxation parameter*. See [113] and [54] for statements and applications of this approach to establish conditions under which collections of adaptive agents who use least squares learning to converge to a rational expectations equilibrium.

Chapter 71

Markov Perfect Equilibrium

71.1 Contents

- Overview 71.2
- Background 71.3
- Linear Markov Perfect Equilibria 71.4
- Application 71.5
- Exercises 71.6
- Solutions 71.7

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

71.2 Overview

This lecture describes the concept of Markov perfect equilibrium.

Markov perfect equilibrium is a key notion for analyzing economic problems involving dynamic strategic interaction, and a cornerstone of applied game theory.

In this lecture, we teach Markov perfect equilibrium by example.

We will focus on settings with

- two players
- quadratic payoff functions
- linear transition rules for the state

Other references include chapter 7 of [108].

Let's start with some standard imports:

```
In [2]: import numpy as np
        import quantecon as qe
        import matplotlib.pyplot as plt
        %matplotlib inline
```

71.3 Background

Markov perfect equilibrium is a refinement of the concept of Nash equilibrium.

It is used to study settings where multiple decision-makers interact non-cooperatively over time, each pursuing its own objective.

The agents in the model face a common state vector, the time path of which is influenced by – and influences – their decisions.

In particular, the transition law for the state that confronts each agent is affected by decision rules of other agents.

Individual payoff maximization requires that each agent solve a dynamic programming problem that includes this transition law.

Markov perfect equilibrium prevails when no agent wishes to revise its policy, taking as given the policies of all other agents.

Well known examples include

- Choice of price, output, location or capacity for firms in an industry (e.g., [52], [137], [47]).
- Rate of extraction from a shared natural resource, such as a fishery (e.g., [107], [158]).

Let's examine a model of the first type.

71.3.1 Example: A Duopoly Model

Two firms are the only producers of a good, the demand for which is governed by a linear inverse demand function

$$p = a_0 - a_1(q_1 + q_2) \tag{1}$$

Here $p = p_t$ is the price of the good, $q_i = q_{it}$ is the output of firm $i = 1, 2$ at time t and $a_0 > 0, a_1 > 0$.

In (1) and what follows,

- the time subscript is suppressed when possible to simplify notation
- $\hat{x}$ denotes a next period value of variable x

Each firm recognizes that its output affects total output and therefore the market price.

The one-period payoff function of firm i is price times quantity minus adjustment costs:

$$\pi_i = pq_i - \gamma(\hat{q}_i - q_i)^2, \quad \gamma > 0, \tag{2}$$

Substituting the inverse demand curve (1) into (2) lets us express the one-period payoff as

$$\pi_i(q_i, q_{-i}, \hat{q}_i) = a_0 q_i - a_1 q_i^2 - a_1 q_i q_{-i} - \gamma(\hat{q}_i - q_i)^2, \tag{3}$$

where q_{-i} denotes the output of the firm other than i.

The objective of the firm is to maximize $\sum_{t=0}^{\infty} \beta^t \pi_{it}$.

Firm i chooses a decision rule that sets next period quantity $\hat{q}_i$ as a function f_i of the current state (q_i, q_{-i}).

An essential aspect of a Markov perfect equilibrium is that each firm takes the decision rule of the other firm as known and given.

Given f_{-i}, the Bellman equation of firm i is

$$v_i(q_i, q_{-i}) = \max_{\hat{q}_i} \{\pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i(\hat{q}_i, f_{-i}(q_{-i}, q_i))\} \tag{4}$$

Definition A *Markov perfect equilibrium* of the duopoly model is a pair of value functions (v_1, v_2) and a pair of policy functions (f_1, f_2) such that, for each $i \in \{1, 2\}$ and each possible state,

- The value function v_i satisfies Bellman equation (4).
- The maximizer on the right side of (4) equals $f_i(q_i, q_{-i})$.

The adjective "Markov" denotes that the equilibrium decision rules depend only on the current values of the state variables, not other parts of their histories.

"Perfect" means complete, in the sense that the equilibrium is constructed by backward induction and hence builds in optimizing behavior for each firm at all possible future states.

- These include many states that will not be reached when we iterate forward on the pair of equilibrium strategies f_i starting from a given initial state.

71.3.2 Computation

One strategy for computing a Markov perfect equilibrium is iterating to convergence on pairs of Bellman equations and decision rules.

In particular, let v_i^j, f_i^j be the value function and policy function for firm i at the j-th iteration.

Imagine constructing the iterates

$$v_i^{j+1}(q_i, q_{-i}) = \max_{\hat{q}_i} \left\{\pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i^j(\hat{q}_i, f_{-i}(q_{-i}, q_i))\right\} \tag{5}$$

These iterations can be challenging to implement computationally.

However, they simplify for the case in which one-period payoff functions are quadratic and transition laws are linear — which takes us to our next topic.

71.4 Linear Markov Perfect Equilibria

As we saw in the duopoly example, the study of Markov perfect equilibria in games with two players leads us to an interrelated pair of Bellman equations.

In linear-quadratic dynamic games, these "stacked Bellman equations" become "stacked Riccati equations" with a tractable mathematical structure.

We'll lay out that structure in a general setup and then apply it to some simple problems.

71.4.1 Coupled Linear Regulator Problems

We consider a general linear-quadratic regulator game with two players.

For convenience, we'll start with a finite horizon formulation, where t_0 is the initial date and t_1 is the common terminal date.

Player i takes $\{u_{-it}\}$ as given and minimizes

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2x_t' W_i u_{it} + 2u_{-it}' M_i u_{it}\} \tag{6}$$

while the state evolves according to

$$x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t} \tag{7}$$

Here

- x_t is an $n \times 1$ state vector and u_{it} is a $k_i \times 1$ vector of controls for player i
- R_i is $n \times n$
- S_i is $k_{-i} \times k_{-i}$
- Q_i is $k_i \times k_i$
- W_i is $n \times k_i$
- M_i is $k_{-i} \times k_i$
- A is $n \times n$
- B_i is $n \times k_i$

71.4.2 Computing Equilibrium

We formulate a linear Markov perfect equilibrium as follows.

Player i employs linear decision rules $u_{it} = -F_{it} x_t$, where F_{it} is a $k_i \times n$ matrix.

A Markov perfect equilibrium is a pair of sequences $\{F_{1t}, F_{2t}\}$ over $t = t_0, \ldots, t_1 - 1$ such that

- $\{F_{1t}\}$ solves player 1's problem, taking $\{F_{2t}\}$ as given, and
- $\{F_{2t}\}$ solves player 2's problem, taking $\{F_{1t}\}$ as given

If we take $u_{2t} = -F_{2t} x_t$ and substitute it into (6) and (7), then player 1's problem becomes minimization of

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' \Pi_{1t} x_t + u_{1t}' Q_1 u_{1t} + 2u_{1t}' \Gamma_{1t} x_t\} \tag{8}$$

subject to

$$x_{t+1} = \Lambda_{1t} x_t + B_1 u_{1t}, \tag{9}$$

where

- $\Lambda_{it} := A - B_{-i} F_{-it}$
- $\Pi_{it} := R_i + F_{-it}' S_i F_{-it}$
- $\Gamma_{it} := W_i' - M_i' F_{-it}$

This is an LQ dynamic programming problem that can be solved by working backwards.

Decision rules that solve this problem are

$$F_{1t} = (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1}(\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) \tag{10}$$

where P_{1t} solves the matrix Riccati difference equation

$$P_{1t} = \Pi_{1t} - (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t})'(Q_1 + \beta B_1' P_{1t+1} B_1)^{-1}(\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) + \beta \Lambda_{1t}' P_{1t+1} \Lambda_{1t} \tag{11}$$

Similarly, decision rules that solve player 2's problem are

$$F_{2t} = (Q_2 + \beta B_2' P_{2t+1} B_2)^{-1}(\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) \tag{12}$$

where P_{2t} solves

$$P_{2t} = \Pi_{2t} - (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t})'(Q_2 + \beta B_2' P_{2t+1} B_2)^{-1}(\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) + \beta \Lambda_{2t}' P_{2t+1} \Lambda_{2t} \tag{13}$$

Here, in all cases $t = t_0, \dots, t_1 - 1$ and the terminal conditions are $P_{it_1} = 0$.

The solution procedure is to use equations (10), (11), (12), and (13), and "work backwards" from time $t_1 - 1$.

Since we're working backward, P_{1t+1} and P_{2t+1} are taken as given at each stage.

Moreover, since

- some terms on the right-hand side of (10) contain F_{2t}
- some terms on the right-hand side of (12) contain F_{1t}

we need to solve these $k_1 + k_2$ equations simultaneously.

Key Insight

A key insight is that equations (10) and (12) are linear in F_{1t} and F_{2t}.

After these equations have been solved, we can take F_{it} and solve for P_{it} in (11) and (13).

Infinite Horizon

We often want to compute the solutions of such games for infinite horizons, in the hope that the decision rules F_{it} settle down to be time-invariant as $t_1 \to +\infty$.

In practice, we usually fix t_1 and compute the equilibrium of an infinite horizon game by driving $t_0 \to -\infty$.

This is the approach we adopt in the next section.

71.4.3 Implementation

We use the function nnash from QuantEcon.py that computes a Markov perfect equilibrium of the infinite horizon linear-quadratic dynamic game in the manner described above.

71.5 Application

Let's use these procedures to treat some applications, starting with the duopoly model.

71.5.1 A Duopoly Model

To map the duopoly model into coupled linear-quadratic dynamic programming problems, define the state and controls as

$$x_t := \begin{bmatrix} 1 \\ q_{1t} \\ q_{2t} \end{bmatrix} \quad \text{and} \quad u_{it} := q_{i,t+1} - q_{it}, \quad i = 1, 2$$

If we write

$$x_t' R_i x_t + u_{it}' Q_i u_{it}$$

where $Q_1 = Q_2 = \gamma$,

$$R_1 := \begin{bmatrix} 0 & -\frac{a_0}{2} & 0 \\ -\frac{a_0}{2} & a_1 & \frac{a_1}{2} \\ 0 & \frac{a_1}{2} & 0 \end{bmatrix} \quad \text{and} \quad R_2 := \begin{bmatrix} 0 & 0 & -\frac{a_0}{2} \\ 0 & 0 & \frac{a_1}{2} \\ -\frac{a_0}{2} & \frac{a_1}{2} & a_1 \end{bmatrix}$$

then we recover the one-period payoffs in expression (3).

The law of motion for the state x_t is $x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t}$ where

$$A := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B_1 := \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad B_2 := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The optimal decision rule of firm i will take the form $u_{it} = -F_i x_t$, inducing the following closed-loop system for the evolution of x in the Markov perfect equilibrium:

$$x_{t+1} = (A - B_1 F_1 - B_1 F_2) x_t \tag{14}$$

71.5.2 Parameters and Solution

Consider the previously presented duopoly model with parameter values of:

- $a_0 = 10$
- $a_1 = 2$
- $\beta = 0.96$
- $\gamma = 12$

From these, we compute the infinite horizon MPE using the preceding code

In [3]:
```
"""
@authors: Chase Coleman, Thomas Sargent, John Stachurski
"""
```

```
import numpy as np
import quantecon as qe

# Parameters
a0 = 10.0
a1 = 2.0
β = 0.96
γ = 12.0

# In LQ form
A = np.eye(3)
B1 = np.array([[0.], [1.], [0.]])
B2 = np.array([[0.], [0.], [1.]])


R1 = [[      0.,      -a0 / 2,           0.],
      [-a0 / 2.,           a1,     a1 / 2.],
      [       0,      a1 / 2.,           0.]]

R2 = [[      0.,            0.,       -a0 / 2],
      [      0.,            0.,       a1 / 2.],
      [-a0 / 2,       a1 / 2.,            a1]]

Q1 = Q2 = γ
S1 = S2 = W1 = W2 = M1 = M2 = 0.0

# Solve using QE's nnash function
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1,
                          Q2, S1, S2, W1, W2, M1,
                          M2, beta=β)

# Display policies
print("Computed policies for firm 1 and firm 2:\n")
print(f"F1 = {F1}")
print(f"F2 = {F2}")
print("\n")
```

```
   Computed policies for firm 1 and firm 2:

F1 = [[-0.66846615  0.29512482  0.07584666]]
F2 = [[-0.66846615  0.07584666  0.29512482]]
```

Running the code produces the following output.

One way to see that F_i is indeed optimal for firm i taking F_2 as given is to use QuantEcon.py's LQ class.

In particular, let's take F2 as computed above, plug it into (8) and (9) to get firm 1's problem and solve it using LQ.

We hope that the resulting policy will agree with F1 as computed above

```
In [4]: Λ1 = A - B2 @ F2
        lq1 = qe.LQ(Q1, R1, Λ1, B1, beta=β)
```

```
P1_ih, F1_ih, d = lq1.stationary_values()
F1_ih
```

```
Out[4]: array([[-0.66846613,  0.29512482,  0.07584666]])
```

This is close enough for rock and roll, as they say in the trade.

Indeed, np.allclose agrees with our assessment

```
In [5]: np.allclose(F1, F1_ih)
```

```
Out[5]: True
```

71.5.3 Dynamics

Let's now investigate the dynamics of price and output in this simple duopoly model under the MPE policies.

Given our optimal policies $F1$ and $F2$, the state evolves according to (14).

The following program

- imports $F1$ and $F2$ from the previous program along with all parameters.
- computes the evolution of x_t using (14).
- extracts and plots industry output $q_t = q_{1t} + q_{2t}$ and price $p_t = a_0 - a_1 q_t$.

```
In [6]: AF = A - B1 @ F1 - B2 @ F2
        n = 20
        x = np.empty((3, n))
        x[:, 0] = 1, 1, 1
        for t in range(n-1):
            x[:, t+1] = AF @ x[:, t]
        q1 = x[1, :]
        q2 = x[2, :]
        q = q1 + q2        # Total output, MPE
        p = a0 - a1 * q    # Price, MPE

        fig, ax = plt.subplots(figsize=(9, 5.8))
        ax.plot(q, 'b-', lw=2, alpha=0.75, label='total output')
        ax.plot(p, 'g-', lw=2, alpha=0.75, label='price')
        ax.set_title('Output and prices, duopoly MPE')
        ax.legend(frameon=False)
        plt.show()
```

Note that the initial condition has been set to $q_{10} = q_{20} = 1.0$.

To gain some perspective we can compare this to what happens in the monopoly case.

The first panel in the next figure compares output of the monopolist and industry output under the MPE, as a function of time.

The second panel shows analogous curves for price.

Here parameters are the same as above for both the MPE and monopoly solutions.

The monopolist initial condition is $q_0 = 2.0$ to mimic the industry initial condition $q_{10} = q_{20} = 1.0$ in the MPE case.

As expected, output is higher and prices are lower under duopoly than monopoly.

71.6 Exercises

71.6.1 Exercise 1

Replicate the pair of figures showing the comparison of output and prices for the monopolist and duopoly under MPE.

Parameters are as in duopoly_mpe.py and you can use that code to compute MPE policies under duopoly.

The optimal policy in the monopolist case can be computed using QuantEcon.py's LQ class.

71.6.2 Exercise 2

In this exercise, we consider a slightly more sophisticated duopoly problem.

It takes the form of infinite horizon linear-quadratic game proposed by Judd [92].

Two firms set prices and quantities of two goods interrelated through their demand curves.

Relevant variables are defined as follows:

- I_{it} = inventories of firm i at beginning of t
- q_{it} = production of firm i during period t
- p_{it} = price charged by firm i during period t
- S_{it} = sales made by firm i during period t
- E_{it} = costs of production of firm i during period t
- C_{it} = costs of carrying inventories for firm i during t

The firms' cost functions are

- $C_{it} = c_{i1} + c_{i2} I_{it} + 0.5 c_{i3} I_{it}^2$
- $E_{it} = e_{i1} + e_{i2} q_{it} + 0.5 e_{i3} q_{it}^2$ where e_{ij}, c_{ij} are positive scalars

Inventories obey the laws of motion

$$I_{i,t+1} = (1-\delta) I_{it} + q_{it} - S_{it}$$

Demand is governed by the linear schedule

$$S_t = D p_{it} + b$$

where

- $S_t = \begin{bmatrix} S_{1t} & S_{2t} \end{bmatrix}'$
- D is a 2×2 negative definite matrix and
- b is a vector of constants

Firm i maximizes the undiscounted sum

$$\lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T} (p_{it} S_{it} - E_{it} - C_{it})$$

We can convert this to a linear-quadratic problem by taking

$$u_{it} = \begin{bmatrix} p_{it} \\ q_{it} \end{bmatrix} \quad \text{and} \quad x_t = \begin{bmatrix} I_{1t} \\ I_{2t} \\ 1 \end{bmatrix}$$

Decision rules for price and quantity take the form $u_{it} = -F_i x_t$.

The Markov perfect equilibrium of Judd's model can be computed by filling in the matrices appropriately.

The exercise is to calculate these matrices and compute the following figures.

The first figure shows the dynamics of inventories for each firm when the parameters are

In [7]:
```
δ = 0.02
D = np.array([[-1, 0.5], [0.5, -1]])
b = np.array([25, 25])
c1 = c2 = np.array([1, -2, 1])
e1 = e2 = np.array([10, 10, 3])
```

Inventories trend to a common steady state.

If we increase the depreciation rate to $\delta = 0.05$, then we expect steady state inventories to fall.

This is indeed the case, as the next figure shows

71.7 Solutions

71.7.1 Exercise 1

First, let's compute the duopoly MPE under the stated parameters

```
In [8]: # == Parameters == #
        a0 = 10.0
        a1 = 2.0
```

```
β = 0.96
γ = 12.0

# == In LQ form == #
A  = np.eye(3)
B1 = np.array([[0.], [1.], [0.]])
B2 = np.array([[0.], [0.], [1.]])
R1 = [[      0.,       -a0/2,           0.],
      [-a0 / 2.,          a1,     a1 / 2.],
      [       0,     a1 / 2.,           0.]]

R2 = [[     0.,           0.,     -a0 / 2],
      [     0.,           0.,     a1 / 2.],
      [-a0 / 2,      a1 / 2.,           a1]]

Q1 = Q2 = γ
S1 = S2 = W1 = W2 = M1 = M2 = 0.0

# == Solve using QE's nnash function == #
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1,
                          Q2, S1, S2, W1, W2, M1,
                          M2, beta=β)
```

Now we evaluate the time path of industry output and prices given initial condition $q_{10} = q_{20} = 1$.

```
In [9]: AF = A - B1 @ F1 - B2 @ F2
        n = 20
        x = np.empty((3, n))
        x[:, 0] = 1, 1, 1
        for t in range(n-1):
            x[:, t+1] = AF @ x[:, t]
        q1 = x[1, :]
        q2 = x[2, :]
        q = q1 + q2        # Total output, MPE
        p = a0 - a1 * q    # Price, MPE
```

Next, let's have a look at the monopoly solution.

For the state and control, we take

$$x_t = q_t - \bar{q} \quad \text{and} \quad u_t = q_{t+1} - q_t$$

To convert to an LQ problem we set

$$R = a_1 \quad \text{and} \quad Q = \gamma$$

in the payoff function $x_t' R x_t + u_t' Q u_t$ and

$$A = B = 1$$

in the law of motion $x_{t+1} = A x_t + B u_t$.

We solve for the optimal policy $u_t = -F x_t$ and track the resulting dynamics of $\{q_t\}$, starting at $q_0 = 2.0$.

In [10]:
```
R = a1
Q = γ
A = B = 1
lq_alt = qe.LQ(Q, R, A, B, beta=β)
P, F, d = lq_alt.stationary_values()
q_bar = a0 / (2.0 * a1)
qm = np.empty(n)
qm[0] = 2
x0 = qm[0] - q_bar
x = x0
for i in range(1, n):
    x = A * x - B * F * x
    qm[i] = float(x) + q_bar
pm = a0 - a1 * qm
```

Let's have a look at the different time paths

In [11]:
```
fig, axes = plt.subplots(2, 1, figsize=(9, 9))

ax = axes[0]
ax.plot(qm, 'b-', lw=2, alpha=0.75, label='monopolist output')
ax.plot(q, 'g-', lw=2, alpha=0.75, label='MPE total output')
ax.set(ylabel="output", xlabel="time", ylim=(2, 4))
ax.legend(loc='upper left', frameon=0)

ax = axes[1]
ax.plot(pm, 'b-', lw=2, alpha=0.75, label='monopolist price')
ax.plot(p, 'g-', lw=2, alpha=0.75, label='MPE price')
ax.set(ylabel="price", xlabel="time")
ax.legend(loc='upper right', frameon=0)
plt.show()
```

71.7.2 Exercise 2

We treat the case $\delta = 0.02$

```
In [12]: δ = 0.02
         D = np.array([[-1, 0.5], [0.5, -1]])
         b = np.array([25, 25])
         c1 = c2 = np.array([1, -2, 1])
         e1 = e2 = np.array([10, 10, 3])

         δ_1 = 1 - δ
```

Recalling that the control and state are

$$u_{it} = \begin{bmatrix} p_{it} \\ q_{it} \end{bmatrix} \quad \text{and} \quad x_t = \begin{bmatrix} I_{1t} \\ I_{2t} \\ 1 \end{bmatrix}$$

we set up the matrices as follows:

```
In [13]: # ==  Create matrices needed to compute the Nash feedback equilibrium == #

         A = np.array([[δ_1,      0,     -δ_1 * b[0]],
                       [  0,    δ_1,     -δ_1 * b[1]],
                       [  0,      0,              1]])

         B1 = δ_1 * np.array([[1, -D[0, 0]],
                              [0, -D[1, 0]],
                              [0,       0]])
         B2 = δ_1 * np.array([[0, -D[0, 1]],
                              [1, -D[1, 1]],
                              [0,       0]])

         R1 = -np.array([[0.5 * c1[2],     0,    0.5 * c1[1]],
                         [          0,     0,              0],
                         [0.5 * c1[1],     0,         c1[0]]])
         R2 = -np.array([[0,               0,              0],
                         [0,     0.5 * c2[2],    0.5 * c2[1]],
                         [0,     0.5 * c2[1],         c2[0]]])

         Q1 = np.array([[-0.5 * e1[2], 0], [0, D[0, 0]]])
         Q2 = np.array([[-0.5 * e2[2], 0], [0, D[1, 1]]])

         S1 = np.zeros((2, 2))
         S2 = np.copy(S1)

         W1 = np.array([[           0,             0],
                        [           0,             0],
                        [-0.5 * e1[1],    b[0] / 2.]])
         W2 = np.array([[           0,             0],
                        [           0,             0],
                        [-0.5 * e2[1],    b[1] / 2.]])

         M1 = np.array([[0, 0], [0, D[0, 1] / 2.]])
         M2 = np.copy(M1)
```

We can now compute the equilibrium using `qe.nnash`

```
In [14]: F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1,
                                   R2, Q1, Q2, S1,
                                   S2, W1, W2, M1, M2)

         print("\nFirm 1's feedback rule:\n")
         print(F1)

         print("\nFirm 2's feedback rule:\n")
         print(F2)
```

```
Firm 1's feedback rule:

[[ 2.43666582e-01  2.72360627e-02 -6.82788293e+00]
 [ 3.92370734e-01  1.39696451e-01 -3.77341073e+01]]

Firm 2's feedback rule:

[[ 2.72360627e-02  2.43666582e-01 -6.82788293e+00]
```

```
[ 1.39696451e-01  3.92370734e-01 -3.77341073e+01]]
```

Now let's look at the dynamics of inventories, and reproduce the graph corresponding to $\delta = 0.02$

```
In [15]: AF = A - B1 @ F1 - B2 @ F2
         n = 25
         x = np.empty((3, n))
         x[:, 0] = 2, 0, 1
         for t in range(n-1):
             x[:, t+1] = AF @ x[:, t]
         I1 = x[0, :]
         I2 = x[1, :]
         fig, ax = plt.subplots(figsize=(9, 5))
         ax.plot(I1, 'b-', lw=2, alpha=0.75, label='inventories, firm 1')
         ax.plot(I2, 'g-', lw=2, alpha=0.75, label='inventories, firm 2')
         ax.set_title(rf'$\delta = {δ}$')
         ax.legend()
         plt.show()
```

Chapter 72

Robust Markov Perfect Equilibrium

72.1 Contents

- Overview 72.2
- Linear Markov Perfect Equilibria with Robust Agents 72.3
- Application 72.4

Co-author: Dongchen Zou

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

72.2 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 [71] and in Markov perfect equilibrium and Robustness.

Let's start with some standard imports:

```
In [2]: import numpy as np
        import quantecon as qe
        from scipy.linalg import solve
        import matplotlib.pyplot as plt
        %matplotlib inline
```

72.2.1 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.

72.3 Linear Markov Perfect Equilibria with Robust Agents

As we saw in Markov perfect equilibrium, the study of Markov perfect equilibria in dynamic games with two players leads us to an interrelated pair of Bellman equations.

In linear quadratic dynamic games, these "stacked Bellman equations" become "stacked Riccati equations" with a tractable mathematical structure.

72.3.1 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

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2x_t' W_i u_{it} + 2u_{-it}' M_i u_{it} - \theta_i v_{it}' v_{it}\right\} \tag{1}$$

while thinking that the state evolves according to

$$x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t} + Cv_{it} \tag{2}$$

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.

72.3.2 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 (1) and (2), then player 1's problem becomes minimization-maximization of

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t'\Pi_{1t}x_t + u_{1t}'Q_1u_{1t} + 2u_{1t}'\Gamma_{1t}x_t - \theta_1 v_{1t}'v_{1t}\} \tag{3}$$

subject to

$$x_{t+1} = \Lambda_{1t}x_t + B_1u_{1t} + Cv_{1t} \tag{4}$$

where

- $\Lambda_{it} := A - B_{-i}F_{-it}$
- $\Pi_{it} := R_i + F_{-it}'S_iF_{-it}$
- $\Gamma_{it} := W_i' - M_i'F_{-it}$

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

$$\mathcal{D}_1(P) := P + PC(\theta_1 I - C'PC)^{-1}C'P \tag{5}$$

The matrix F_{1t} in the policy rule $u_{1t} = -F_{1t}x_t$ that solves agent 1's problem satisfies

$$F_{1t} = (Q_1 + \beta B_1'\mathcal{D}_1(P_{1t+1})B_1)^{-1}(\beta B_1'\mathcal{D}_1(P_{1t+1})\Lambda_{1t} + \Gamma_{1t}) \tag{6}$$

where P_{1t} solves the matrix Riccati difference equation

$$P_{1t} = \Pi_{1t} - (\beta B_1'\mathcal{D}_1(P_{1t+1})\Lambda_{1t} + \Gamma_{1t})'(Q_1 + \beta B_1'\mathcal{D}_1(P_{1t+1})B_1)^{-1}(\beta B_1'\mathcal{D}_1(P_{1t+1})\Lambda_{1t} + \Gamma_{1t}) + \beta\Lambda_{1t}'\mathcal{D}_1(P_{1t+1})\Lambda_{1t} \tag{7}$$

Similarly, the policy that solves player 2's problem is

$$F_{2t} = (Q_2 + \beta B_2'\mathcal{D}_2(P_{2t+1})B_2)^{-1}(\beta B_2'\mathcal{D}_2(P_{2t+1})\Lambda_{2t} + \Gamma_{2t}) \tag{8}$$

where P_{2t} solves

$$P_{2t} = \Pi_{2t} - (\beta B_2'\mathcal{D}_2(P_{2t+1})\Lambda_{2t} + \Gamma_{2t})'(Q_2 + \beta B_2'\mathcal{D}_2(P_{2t+1})B_2)^{-1}(\beta B_2'\mathcal{D}_2(P_{2t+1})\Lambda_{2t} + \Gamma_{2t}) + \beta\Lambda_{2t}'\mathcal{D}_2(P_{2t+1})\Lambda_{2t} \tag{9}$$

Here in all cases $t = t_0, \dots, t_1 - 1$ and the terminal conditions are $P_{it_1} = 0$.

The solution procedure is to use equations (6), (7), (8), and (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 (6) contain F_{2t}
- some terms on the right-hand side of (8) contain F_{1t}

we need to solve these $k_1 + k_2$ equations simultaneously.

72.3.3 Key Insight

As in Markov perfect equilibrium, a key insight here is that equations (6) and (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 (7) and (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 $\{F_{it}; t = t_0, \dots, t_1 - 1\}$ 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.

72.3.4 Worst-case Shocks

For agent i the maximizing or worst-case shock v_{it} is

$$v_{it} = K_{it} x_t$$

where

$$K_{it} = \theta_i^{-1}(I - \theta_i^{-1} C' P_{i,t+1} C)^{-1} C' P_{i,t+1}(A - B_1 F_{it} - B_2 F_{2t})$$

72.3.5 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.

72.3.6 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.

72.4 Application

72.4.1 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

$$p = a_0 - a_1(q_1 + q_2) \tag{10}$$

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 (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:

$$\pi_i = pq_i - \gamma(\hat{q}_i - q_i)^2, \quad \gamma > 0, \tag{11}$$

Substituting the inverse demand curve (10) into (11) lets us express the one-period payoff as

$$\pi_i(q_i, q_{-i}, \hat{q}_i) = a_0 q_i - a_1 q_i^2 - a_1 q_i q_{-i} - \gamma(\hat{q}_i - q_i)^2, \tag{12}$$

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 (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:

$$x_{t+1} = (A - B_1 F_1 - B_1 F_2) x_t \tag{13}$$

72.4.2 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

In [3]:
```
"""
@authors: Chase Coleman, Thomas Sargent, John Stachurski
"""

import numpy as np
import quantecon as qe

# Parameters
a0 = 10.0
a1 = 2.0
β = 0.96
γ = 12.0

# In LQ form
A = np.eye(3)
B1 = np.array([[0.], [1.], [0.]])
B2 = np.array([[0.], [0.], [1.]])


R1 = [[      0.,      -a0 / 2,           0.],
      [-a0 / 2.,           a1,     a1 / 2.],
      [       0,      a1 / 2.,           0.]]

R2 = [[      0.,            0.,       -a0 / 2],
      [      0.,            0.,       a1 / 2.],
      [-a0 / 2,       a1 / 2.,            a1]]
```

```
Q1 = Q2 = γ
S1 = S2 = W1 = W2 = M1 = M2 = 0.0

# Solve using QE's nnash function
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1,
                          Q2, S1, S2, W1, W2, M1,
                          M2, beta=β)

# 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

```
In [4]: 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}
                \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)
            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

    # 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 {maxiter} \
            reached in nnash')

    return F1, F2, P1, P2
```

72.4.3 Some Details

Firm i wants to minimize

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2x_t' W_i u_{it} + 2u_{-it}' M_i u_{it}\}$$

where

$$x_t := \begin{bmatrix} 1 \\ q_{1t} \\ q_{2t} \end{bmatrix} \quad \text{and} \quad u_{it} := q_{i,t+1} - q_{it}, \quad i = 1, 2$$

and

$$R_1 := \begin{bmatrix} 0 & -\frac{a_0}{2} & 0 \\ -\frac{a_0}{2} & a_1 & \frac{a_1}{2} \\ 0 & \frac{a_1}{2} & 0 \end{bmatrix}, \quad R_2 := \begin{bmatrix} 0 & 0 & -\frac{a_0}{2} \\ 0 & 0 & \frac{a_1}{2} \\ -\frac{a_0}{2} & \frac{a_1}{2} & a_1 \end{bmatrix}, \quad Q_1 = Q_2 = \gamma, \quad S_1 = S_2 = 0, \quad W_1 = W_2 = 0, \quad M_1 =$$

The parameters of the duopoly model are:

- $a_0 = 10$
- $a_1 = 2$
- $\beta = 0.96$
- $\gamma = 12$

In [5]:
```
# 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 $\theta_i \approx +\infty$

In [6]:
```
# 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))
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.

In [7]:
```
# Robustness parameters and matrix
C = np.asmatrix([[0], [0.01], [0.01]])
θ1 = 0.02
θ2 = 0.04
n = 20


# 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 base-

line model; only the decision rules F_i differ across the two equilibrium objects presented.

```
In [8]: 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

In [9]:
```
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')
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

```
In [10]: 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]]
```

In [11]:
```
# == 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)

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.

([71] discuss how this property of robust decision rules is connected to the concept of *admissibility* in Bayesian statistical decision theory)

Chapter 73

Uncertainty Traps

73.1 Contents

- Overview 73.2
- The Model 73.3
- Implementation 73.4
- Results 73.5
- Exercises 73.6
- Solutions 73.7

73.2 Overview

In this lecture, we study a simplified version of an uncertainty traps model of Fajgelbaum, Schaal and Taschereau-Dumouchel [55].

The model features self-reinforcing uncertainty that has big impacts on economic activity.

In the model,

- Fundamentals vary stochastically and are not fully observable.
- At any moment there are both active and inactive entrepreneurs; only active entrepreneurs produce.
- Agents – active and inactive entrepreneurs – have beliefs about the fundamentals expressed as probability distributions.
- Greater uncertainty means greater dispersions of these distributions.
- Entrepreneurs are risk-averse and hence less inclined to be active when uncertainty is high.
- The output of active entrepreneurs is observable, supplying a noisy signal that helps everyone inside the model infer fundamentals.
- Entrepreneurs update their beliefs about fundamentals using Bayes' Law, implemented via Kalman filtering.

Uncertainty traps emerge because:

- High uncertainty discourages entrepreneurs from becoming active.
- A low level of participation – i.e., a smaller number of active entrepreneurs – diminishes the flow of information about fundamentals.
- Less information translates to higher uncertainty, further discouraging entrepreneurs

from choosing to be active, and so on.

Uncertainty traps stem from a positive externality: high aggregate economic activity levels generates valuable information.

Let's start with some standard imports:

```
In [1]: import matplotlib.pyplot as plt
        %matplotlib inline
        import numpy as np
        import itertools
```

73.3 The Model

The original model described in [55] has many interesting moving parts.

Here we examine a simplified version that nonetheless captures many of the key ideas.

73.3.1 Fundamentals

The evolution of the fundamental process $\{\theta_t\}$ is given by

$$\theta_{t+1} = \rho\theta_t + \sigma_\theta w_{t+1}$$

where

- $\sigma_\theta > 0$ and $0 < \rho < 1$
- $\{w_t\}$ is IID and standard normal

The random variable θ_t is not observable at any time.

73.3.2 Output

There is a total $\bar{M}$ of risk-averse entrepreneurs.

Output of the m-th entrepreneur, conditional on being active in the market at time t, is equal to

$$x_m = \theta + \epsilon_m \quad \text{where} \quad \epsilon_m \sim N\left(0, \gamma_x^{-1}\right) \tag{1}$$

Here the time subscript has been dropped to simplify notation.

The inverse of the shock variance, γ_x, is called the shock's **precision**.

The higher is the precision, the more informative x_m is about the fundamental.

Output shocks are independent across time and firms.

73.3.3 Information and Beliefs

All entrepreneurs start with identical beliefs about θ_0.

Signals are publicly observable and hence all agents have identical beliefs always.

Dropping time subscripts, beliefs for current θ are represented by the normal distribution $N(\mu, \gamma^{-1})$.

Here γ is the precision of beliefs; its inverse is the degree of uncertainty.

These parameters are updated by Kalman filtering.

Let

- $\mathbb{M} \subset \{1, \dots, \bar{M}\}$ denote the set of currently active firms.
- $M := |\mathbb{M}|$ denote the number of currently active firms.
- X be the average output $\frac{1}{M} \sum_{m \in \mathbb{M}} x_m$ of the active firms.

With this notation and primes for next period values, we can write the updating of the mean and precision via

$$\mu' = \rho \frac{\gamma \mu + M \gamma_x X}{\gamma + M \gamma_x} \tag{2}$$

$$\gamma' = \left(\frac{\rho^2}{\gamma + M \gamma_x} + \sigma_\theta^2 \right)^{-1} \tag{3}$$

These are standard Kalman filtering results applied to the current setting.

Exercise 1 provides more details on how (2) and (3) are derived and then asks you to fill in remaining steps.

The next figure plots the law of motion for the precision in (3) as a 45 degree diagram, with one curve for each $M \in \{0, \dots, 6\}$.

The other parameter values are $\rho = 0.99, \gamma_x = 0.5, \sigma_\theta = 0.5$

Points where the curves hit the 45 degree lines are long-run steady states for precision for different values of M.

Thus, if one of these values for M remains fixed, a corresponding steady state is the equilibrium level of precision

- high values of M correspond to greater information about the fundamental, and hence more precision in steady state
- low values of M correspond to less information and more uncertainty in steady state

In practice, as we'll see, the number of active firms fluctuates stochastically.

73.3.4 Participation

Omitting time subscripts once more, entrepreneurs enter the market in the current period if

$$\mathbb{E}[u(x_m - F_m)] > c \tag{4}$$

Here

- the mathematical expectation of x_m is based on (1) and beliefs $N(\mu, \gamma^{-1})$ for θ
- F_m is a stochastic but pre-visible fixed cost, independent across time and firms
- c is a constant reflecting opportunity costs

The statement that F_m is pre-visible means that it is realized at the start of the period and

treated as a constant in (4).

The utility function has the constant absolute risk aversion form

$$u(x) = \frac{1}{a}\left(1 - \exp(-ax)\right) \tag{5}$$

where a is a positive parameter.

Combining (4) and (5), entrepreneur m participates in the market (or is said to be active) when

$$\frac{1}{a}\left\{1 - \mathbb{E}[\exp\left(-a(\theta + \epsilon_m - F_m)\right)]\right\} > c$$

Using standard formulas for expectations of lognormal random variables, this is equivalent to the condition

$$\psi(\mu, \gamma, F_m) := \frac{1}{a}\left(1 - \exp\left(-a\mu + aF_m + \frac{a^2\left(\frac{1}{\gamma} + \frac{1}{\gamma_x}\right)}{2}\right)\right) - c > 0 \tag{6}$$

73.4 Implementation

We want to simulate this economy.

As a first step, let's put together a class that bundles

- the parameters, the current value of θ and the current values of the two belief parameters μ and γ
- methods to update θ, μ and γ, as well as to determine the number of active firms and their outputs

The updating methods follow the laws of motion for θ, μ and γ given above.

The method to evaluate the number of active firms generates $F_1, \dots, F_{\bar{M}}$ and tests condition (6) for each firm.

The **init** method encodes as default values the parameters we'll use in the simulations below

```
In [2]: class UncertaintyTrapEcon:

            def __init__(self,
                         a=1.5,            # Risk aversion
                         γ_x=0.5,          # Production shock precision
                         ρ=0.99,           # Correlation coefficient for ϑ
                         σ_θ=0.5,          # Standard dev of ϑ shock
                         num_firms=100,    # Number of firms
                         σ_F=1.5,          # Standard dev of fixed costs
                         c=-420,           # External opportunity cost
                         μ_init=0,         # Initial value for μ
                         γ_init=4,         # Initial value for γ
                         θ_init=0):        # Initial value for ϑ

                # == Record values == #
                self.a, self.γ_x, self.ρ, self.σ_θ = a, γ_x, ρ, σ_θ
```

```
        self.num_firms, self.σ_F, self.c, = num_firms, σ_F, c
        self.σ_x = np.sqrt(1/γ_x)

        # == Initialize states == #
        self.γ, self.μ, self.θ = γ_init, μ_init, θ_init

    def ψ(self, F):
        temp1 = -self.a * (self.μ - F)
        temp2 = self.a**2 * (1/self.γ + 1/self.γ_x) / 2
        return (1 / self.a) * (1 - np.exp(temp1 + temp2)) - self.c

    def update_beliefs(self, X, M):
        """
        Update beliefs (μ, γ) based on aggregates X and M.
        """
        # Simplify names
        γ_x, ρ, σ_θ = self.γ_x, self.ρ, self.σ_θ
        # Update μ
        temp1 = ρ * (self.γ * self.μ + M * γ_x * X)
        temp2 = self.γ + M * γ_x
        self.μ = temp1 / temp2
        # Update γ
        self.γ = 1 / (ρ**2 / (self.γ + M * γ_x) + σ_θ**2)

    def update_θ(self, w):
        """
        Update the fundamental state θ given shock w.
        """
        self.θ = self.ρ * self.θ + self.σ_θ * w

    def gen_aggregates(self):
        """
        Generate aggregates based on current beliefs (μ, γ). This
        is a simulation step that depends on the draws for F.
        """
        F_vals = self.σ_F * np.random.randn(self.num_firms)
        M = np.sum(self.ψ(F_vals) > 0)  # Counts number of active firms
        if M > 0:
            x_vals = self.θ + self.σ_x * np.random.randn(M)
            X = x_vals.mean()
        else:
            X = 0
        return X, M
```

In the results below we use this code to simulate time series for the major variables.

73.5 Results

Let's look first at the dynamics of μ, which the agents use to track θ

We see that μ tracks θ well when there are sufficient firms in the market.

However, there are times when μ tracks θ poorly due to insufficient information.

These are episodes where the uncertainty traps take hold.

During these episodes

- precision is low and uncertainty is high
- few firms are in the market

To get a clearer idea of the dynamics, let's look at all the main time series at once, for a given set of shocks

Notice how the traps only take hold after a sequence of bad draws for the fundamental.

Thus, the model gives us a *propagation mechanism* that maps bad random draws into long downturns in economic activity.

73.6 Exercises

73.6.1 Exercise 1

Fill in the details behind (2) and (3) based on the following standard result (see, e.g., p. 24 of [166]).

Fact Let $\mathbf{x} = (x_1, \dots, x_M)$ be a vector of IID draws from common distribution $N(\theta, 1/\gamma_x)$ and let $\bar{x}$ be the sample mean. If γ_x is known and the prior for θ is $N(\mu, 1/\gamma)$, then the posterior distribution of θ given $\mathbf{x}$ is

$$\pi(\theta \,|\, \mathbf{x}) = N(\mu_0, 1/\gamma_0)$$

where

$$\mu_0 = \frac{\mu\gamma + M\bar{x}\gamma_x}{\gamma + M\gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M\gamma_x$$

73.6.2 Exercise 2

Modulo randomness, replicate the simulation figures shown above.

- Use the parameter values listed as defaults in the **init** method of the UncertaintyTrapEcon class.

73.7 Solutions

73.7.1 Exercise 1

This exercise asked you to validate the laws of motion for γ and μ given in the lecture, based on the stated result about Bayesian updating in a scalar Gaussian setting. The stated result tells us that after observing average output X of the M firms, our posterior beliefs will be

$$N(\mu_0, 1/\gamma_0)$$

where

$$\mu_0 = \frac{\mu\gamma + MX\gamma_x}{\gamma + M\gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M\gamma_x$$

If we take a random variable θ with this distribution and then evaluate the distribution of $\rho\theta + \sigma_\theta w$ where w is independent and standard normal, we get the expressions for μ' and γ' given in the lecture.

73.7.2 Exercise 2

First, let's replicate the plot that illustrates the law of motion for precision, which is

$$\gamma_{t+1} = \left(\frac{\rho^2}{\gamma_t + M\gamma_x} + \sigma_\theta^2 \right)^{-1}$$

Here M is the number of active firms. The next figure plots γ_{t+1} against γ_t on a 45 degree diagram for different values of M

In [3]:
```
econ = UncertaintyTrapEcon()
ρ, σ_θ, γ_x = econ.ρ, econ.σ_θ, econ.γ_x     # Simplify names
γ = np.linspace(1e-10, 3, 200)               # γ grid
fig, ax = plt.subplots(figsize=(9, 9))
ax.plot(γ, γ, 'k-')                          # 45 degree line

for M in range(7):
    γ_next = 1 / (ρ**2 / (γ + M * γ_x) + σ_θ**2)
    label_string = f"$M = {M}$"
    ax.plot(γ, γ_next, lw=2, label=label_string)
ax.legend(loc='lower right', fontsize=14)
ax.set_xlabel(r'$\gamma$', fontsize=16)
ax.set_ylabel(r"$\gamma'$", fontsize=16)
ax.grid()
plt.show()
```

The points where the curves hit the 45 degree lines are the long-run steady states corresponding to each M, if that value of M was to remain fixed. As the number of firms falls, so does the long-run steady state of precision.

Next let's generate time series for beliefs and the aggregates – that is, the number of active firms and average output

In [4]:
```
sim_length=2000

μ_vec = np.empty(sim_length)
θ_vec = np.empty(sim_length)
γ_vec = np.empty(sim_length)
X_vec = np.empty(sim_length)
M_vec = np.empty(sim_length)

μ_vec[0] = econ.μ
γ_vec[0] = econ.γ
θ_vec[0] = 0

w_shocks = np.random.randn(sim_length)

for t in range(sim_length-1):
    X, M = econ.gen_aggregates()
    X_vec[t] = X
    M_vec[t] = M

    econ.update_beliefs(X, M)
    econ.update_θ(w_shocks[t])

    μ_vec[t+1] = econ.μ
    γ_vec[t+1] = econ.γ
    θ_vec[t+1] = econ.θ

# Record final values of aggregates
X, M = econ.gen_aggregates()
X_vec[-1] = X
M_vec[-1] = M
```

First, let's see how well μ tracks θ in these simulations

In [5]:
```
fig, ax = plt.subplots(figsize=(9, 6))
ax.plot(range(sim_length), θ_vec, alpha=0.6, lw=2, label=r"$\theta$")
ax.plot(range(sim_length), μ_vec, alpha=0.6, lw=2, label=r"$\mu$")
ax.legend(fontsize=16)
ax.grid()
plt.show()
```

Now let's plot the whole thing together

```
In [6]: fig, axes = plt.subplots(4, 1, figsize=(12, 20))
        # Add some spacing
        fig.subplots_adjust(hspace=0.3)

        series = (θ_vec, μ_vec, γ_vec, M_vec)
        names = r'$\theta$', r'$\mu$', r'$\gamma$', r'$M$'

        for ax, vals, name in zip(axes, series, names):
            # Determine suitable y limits
            s_max, s_min = max(vals), min(vals)
            s_range = s_max - s_min
            y_max = s_max + s_range * 0.1
            y_min = s_min - s_range * 0.1
            ax.set_ylim(y_min, y_max)
            # Plot series
            ax.plot(range(sim_length), vals, alpha=0.6, lw=2)
            ax.set_title(f"time series for {name}", fontsize=16)
            ax.grid()

        plt.show()
```

If you run the code above you'll get different plots, of course.

Try experimenting with different parameters to see the effects on the time series.

(It would also be interesting to experiment with non-Gaussian distributions for the shocks, but this is a big exercise since it takes us outside the world of the standard Kalman filter)

Chapter 74

The Aiyagari Model

74.1 Contents

- Overview 74.2
- The Economy 74.3
- Firms 74.4
- Code 74.5

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

74.2 Overview

In this lecture, we describe the structure of a class of models that build on work by Truman Bewley [21].

We begin by discussing an example of a Bewley model due to Rao Aiyagari.

The model features

- Heterogeneous agents
- A single exogenous vehicle for borrowing and lending
- Limits on amounts individual agents may borrow

The Aiyagari model has been used to investigate many topics, including

- precautionary savings and the effect of liquidity constraints [6]
- risk sharing and asset pricing [82]
- the shape of the wealth distribution [18]
- etc., etc., etc.

Let's start with some imports:

```
In [2]: import numpy as np
        import quantecon as qe
        import matplotlib.pyplot as plt
        %matplotlib inline
        from quantecon.markov import DiscreteDP
        from numba import jit
```

74.2.1 References

The primary reference for this lecture is [6].

A textbook treatment is available in chapter 18 of [108].

A continuous time version of the model by SeHyoun Ahn and Benjamin Moll can be found here.

74.3 The Economy

74.3.1 Households

Infinitely lived households / consumers face idiosyncratic income shocks.

A unit interval of *ex-ante* identical households face a common borrowing constraint.

The savings problem faced by a typical household is

$$\max \mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$a_{t+1} + c_t \leq w z_t + (1+r) a_t \quad c_t \geq 0, \quad \text{and} \quad a_t \geq -B$$

where

- c_t is current consumption
- a_t is assets
- z_t is an exogenous component of labor income capturing stochastic unemployment risk, etc.
- w is a wage rate
- r is a net interest rate
- B is the maximum amount that the agent is allowed to borrow

The exogenous process $\{z_t\}$ follows a finite state Markov chain with given stochastic matrix P.

The wage and interest rate are fixed over time.

In this simple version of the model, households supply labor inelastically because they do not value leisure.

74.4 Firms

Firms produce output by hiring capital and labor.

Firms act competitively and face constant returns to scale.

Since returns to scale are constant the number of firms does not matter.

Hence we can consider a single (but nonetheless competitive) representative firm.

The firm's output is

$$Y_t = AK_t^\alpha N^{1-\alpha}$$

where

- A and α are parameters with $A > 0$ and $\alpha \in (0, 1)$
- K_t is aggregate capital
- N is total labor supply (which is constant in this simple version of the model)

The firm's problem is

$$max_{K,N} \left\{ AK_t^\alpha N^{1-\alpha} - (r + \delta)K - wN \right\}$$

The parameter δ is the depreciation rate.

From the first-order condition with respect to capital, the firm's inverse demand for capital is

$$r = A\alpha \left(\frac{N}{K} \right)^{1-\alpha} - \delta \tag{1}$$

Using this expression and the firm's first-order condition for labor, we can pin down the equilibrium wage rate as a function of r as

$$w(r) = A(1 - \alpha)(A\alpha/(r + \delta))^{\alpha/(1-\alpha)} \tag{2}$$

74.4.1 Equilibrium

We construct a *stationary rational expectations equilibrium* (SREE).

In such an equilibrium

- prices induce behavior that generates aggregate quantities consistent with the prices
- aggregate quantities and prices are constant over time

In more detail, an SREE lists a set of prices, savings and production policies such that

- households want to choose the specified savings policies taking the prices as given
- firms maximize profits taking the same prices as given
- the resulting aggregate quantities are consistent with the prices; in particular, the demand for capital equals the supply
- aggregate quantities (defined as cross-sectional averages) are constant

In practice, once parameter values are set, we can check for an SREE by the following steps

1. pick a proposed quantity K for aggregate capital
2. determine corresponding prices, with interest rate r determined by (1) and a wage rate $w(r)$ as given in (2)
3. determine the common optimal savings policy of the households given these prices
4. compute aggregate capital as the mean of steady state capital given this savings policy

If this final quantity agrees with K then we have a SREE.

74.5 Code

Let's look at how we might compute such an equilibrium in practice.

To solve the household's dynamic programming problem we'll use the DiscreteDP class from QuantEcon.py.

Our first task is the least exciting one: write code that maps parameters for a household problem into the `R` and `Q` matrices needed to generate an instance of `DiscreteDP`.

Below is a piece of boilerplate code that does just this.

In reading the code, the following information will be helpful

- `R` needs to be a matrix where `R[s, a]` is the reward at state `s` under action `a`.
- `Q` needs to be a three-dimensional array where `Q[s, a, s']` is the probability of transitioning to state `s'` when the current state is `s` and the current action is `a`.

(For a detailed discussion of `DiscreteDP` see this lecture)

Here we take the state to be $s_t := (a_t, z_t)$, where a_t is assets and z_t is the shock.

The action is the choice of next period asset level a_{t+1}.

We use Numba to speed up the loops so we can update the matrices efficiently when the parameters change.

The class also includes a default set of parameters that we'll adopt unless otherwise specified.

```
In [3]: class Household:
            """
            This class takes the parameters that define a household asset
↪accumulation
           problem and computes the corresponding reward and transition matrices R
           and Q required to generate an instance of DiscreteDP, and thereby solve
           for the optimal policy.

           Comments on indexing: We need to enumerate the state space S as a
↪sequence
           S = {0, ..., n}.  To this end, (a_i, z_i) index pairs are mapped to s_i
           indices according to the rule

               s_i = a_i * z_size + z_i

           To invert this map, use

               a_i = s_i // z_size  (integer division)
               z_i = s_i % z_size

           """


           def __init__(self,
                        r=0.01,                       # Interest rate
                        w=1.0,                        # Wages
                        β=0.96,                       # Discount factor
                        a_min=1e-10,
                        Π=[[0.9, 0.1], [0.1, 0.9]],   # Markov chain
                        z_vals=[0.1, 1.0],            # Exogenous states
                        a_max=18,
```

```
                a_size=200):

        # Store values, set up grids over a and z
        self.r, self.w, self.β = r, w, β
        self.a_min, self.a_max, self.a_size = a_min, a_max, a_size

        self.Π = np.asarray(Π)
        self.z_vals = np.asarray(z_vals)
        self.z_size = len(z_vals)

        self.a_vals = np.linspace(a_min, a_max, a_size)
        self.n = a_size * self.z_size

        # Build the array Q
        self.Q = np.zeros((self.n, a_size, self.n))
        self.build_Q()

        # Build the array R
        self.R = np.empty((self.n, a_size))
        self.build_R()

    def set_prices(self, r, w):
        """
        Use this method to reset prices. Calling the method will trigger a
        re-build of R.
        """
        self.r, self.w = r, w
        self.build_R()

    def build_Q(self):
        populate_Q(self.Q, self.a_size, self.z_size, self.Π)

    def build_R(self):
        self.R.fill(-np.inf)
        populate_R(self.R,
                self.a_size,
                self.z_size,
                self.a_vals,
                self.z_vals,
                self.r,
                self.w)


# Do the hard work using JIT-ed functions

@jit(nopython=True)
def populate_R(R, a_size, z_size, a_vals, z_vals, r, w):
    n = a_size * z_size
    for s_i in range(n):
        a_i = s_i // z_size
        z_i = s_i % z_size
        a = a_vals[a_i]
        z = z_vals[z_i]
        for new_a_i in range(a_size):
            a_new = a_vals[new_a_i]
            c = w * z + (1 + r) * a - a_new
            if c > 0:
                R[s_i, new_a_i] = np.log(c)  # Utility
```

```
@jit(nopython=True)
def populate_Q(Q, a_size, z_size, Π):
    n = a_size * z_size
    for s_i in range(n):
        z_i = s_i % z_size
        for a_i in range(a_size):
            for next_z_i in range(z_size):
                Q[s_i, a_i, a_i*z_size + next_z_i] = Π[z_i, next_z_i]


@jit(nopython=True)
def asset_marginal(s_probs, a_size, z_size):
    a_probs = np.zeros(a_size)
    for a_i in range(a_size):
        for z_i in range(z_size):
            a_probs[a_i] += s_probs[a_i*z_size + z_i]
    return a_probs
```

As a first example of what we can do, let's compute and plot an optimal accumulation policy at fixed prices.

In [4]:
```
# Example prices
r = 0.03
w = 0.956

# Create an instance of Household
am = Household(a_max=20, r=r, w=w)

# Use the instance to build a discrete dynamic program
am_ddp = DiscreteDP(am.R, am.Q, am.β)

# Solve using policy function iteration
results = am_ddp.solve(method='policy_iteration')

# Simplify names
z_size, a_size = am.z_size, am.a_size
z_vals, a_vals = am.z_vals, am.a_vals
n = a_size * z_size

# Get all optimal actions across the set of a indices with z fixed in each
↪row
a_star = np.empty((z_size, a_size))
for s_i in range(n):
    a_i = s_i // z_size
    z_i = s_i % z_size
    a_star[z_i, a_i] = a_vals[results.sigma[s_i]]

fig, ax = plt.subplots(figsize=(9, 9))
ax.plot(a_vals, a_vals, 'k--')  # 45 degrees
for i in range(z_size):
    lb = f'$z = {z_vals[i]:.2}$'
    ax.plot(a_vals, a_star[i, :], lw=2, alpha=0.6, label=lb)
    ax.set_xlabel('current assets')
    ax.set_ylabel('next period assets')
ax.legend(loc='upper left')

plt.show()
```

The plot shows asset accumulation policies at different values of the exogenous state.

Now we want to calculate the equilibrium.

Let's do this visually as a first pass.

The following code draws aggregate supply and demand curves.

The intersection gives equilibrium interest rates and capital.

In [5]:
```
A = 1.0
N = 1.0
α = 0.33
β = 0.96
δ = 0.05


def r_to_w(r):
    """
    Equilibrium wages associated with a given interest rate r.
    """
    return A * (1 - α) * (A * α / (r + δ))**(α / (1 - α))

def rd(K):
```

```
    """
    Inverse demand curve for capital.  The interest rate associated with a
    given demand for capital K.
    """
    return A * α * (N / K)**(1 - α) - δ


def prices_to_capital_stock(am, r):
    """
    Map prices to the induced level of capital stock.

    Parameters:
    ----------

    am : Household
        An instance of an aiyagari_household.Household
    r : float
        The interest rate
    """
    w = r_to_w(r)
    am.set_prices(r, w)
    aiyagari_ddp = DiscreteDP(am.R, am.Q, β)
    # Compute the optimal policy
    results = aiyagari_ddp.solve(method='policy_iteration')
    # Compute the stationary distribution
    stationary_probs = results.mc.stationary_distributions[0]
    # Extract the marginal distribution for assets
    asset_probs = asset_marginal(stationary_probs, am.a_size, am.z_size)
    # Return K
    return np.sum(asset_probs * am.a_vals)


# Create an instance of Household
am = Household(a_max=20)

# Use the instance to build a discrete dynamic program
am_ddp = DiscreteDP(am.R, am.Q, am.β)

# Create a grid of r values at which to compute demand and supply of capital
num_points = 20
r_vals = np.linspace(0.005, 0.04, num_points)

# Compute supply of capital
k_vals = np.empty(num_points)
for i, r in enumerate(r_vals):
    k_vals[i] = prices_to_capital_stock(am, r)

# Plot against demand for capital by firms
fig, ax = plt.subplots(figsize=(11, 8))
ax.plot(k_vals, r_vals, lw=2, alpha=0.6, label='supply of capital')
ax.plot(k_vals, rd(k_vals), lw=2, alpha=0.6, label='demand for capital')
ax.grid()
ax.set_xlabel('capital')
ax.set_ylabel('interest rate')
ax.legend(loc='upper right')

plt.show()
```

supply of capital
demand for capital
0.08
0.06
0.04
0.02
interest rate
4
6
8
10
12
capital

Chapter 75

Default Risk and Income Fluctuations

75.1 Contents

- Overview 75.2
- Structure 75.3
- Equilibrium 75.4
- Computation 75.5
- Results 75.6
- Exercises 75.7
- Solutions 75.8

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

75.2 Overview

This lecture computes versions of Arellano's [10] 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:

In [2]:
```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
import random

from numba import jit, jitclass, int64, float64

%matplotlib inline
```

75.3 Structure

In this section we describe the main features of the model.

75.3.1 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{1}$$

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 (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.

75.3.2 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

$$c = y + B - q(B', y)B' \tag{2}$$

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.

75.3.3 Financial Markets

Foreign creditors

- are risk neutral
- know the domestic output stochastic process $\{y_t\}$ and observe $y_t, y_{t-1}, \dots$, at time t
- can borrow or lend without limit in an international credit market at a constant international interest rate r
- receive full payment if the government chooses to pay
- receive zero if the government defaults on its one-period debt due

When a government is expected to default next period with probability δ, the expected value of a promise to pay one unit of consumption next period is $1 - \delta$.

Therefore, the discounted expected value of a promise to pay B next period is

$$q = \frac{1 - \delta}{1 + r} \tag{3}$$

Next we turn to how the government in effect chooses the default probability δ.

75.3.4 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.

1. The country loses access to foreign credit markets.

75.3.5 Reentering International Credit Market

While in a state of default, the economy regains access to foreign credit in each subsequent period with probability θ.

75.4 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

1. 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

4. to default, or

5. 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

$$\delta(B', y) := \int \mathbb{1}\{v_c(B', y') < v_d(y')\} p(y, y') dy' \tag{4}$$

Given zero profits for foreign creditors in equilibrium, we can combine (3) and (4) to pin down the bond price function:

$$q(B', y) = \frac{1 - \delta(B', y)}{1 + r} \tag{5}$$

75.4.1 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 (5)

75.5 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 [10] 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 $v_c(B, y)$.

1. Update the value function $v(B, y)$, the default rule, the implied ex ante default probability, and the price function.

2. 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.)

In [3]:
```
# 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

In [4]:
```
@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
    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)

        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

```
In [5]: @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.ones((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
                    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

```
In [6]: 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
            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
```

75.6 Results

Let's start by trying to replicate the results obtained in [10].

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.

The next figure plots value functions and replicates the right hand panel of Figure 4 of [10].

We can use the results of the computation to study the default probability $\delta(B', y)$ defined in (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.

75.7 Exercises

75.7.1 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.

75.8 Solutions

Compute the value function, policy and equilibrium prices

In [7]:
```
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
)
```

In [8]:
```
V, Vc, Vd, iBstar, default_prob, default_states, q = solve(ae)
```

Compute the bond price schedule as seen in figure 3 of Arellano (2008)

In [9]:
```
# 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

```
In [10]: # 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

```
In [11]: 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()
```

Plot a time series of major variables simulated from the model

```
In [12]: 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
             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()
```

Chapter 76

Globalization and Cycles

76.1 Contents

- Overview 76.2
- Key Ideas 76.3
- Model 76.4
- Simulation 76.5
- Exercises 76.6
- Solutions 76.7

This lecture is coauthored with Chase Coleman.

76.2 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:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
from numba import jit, vectorize
from ipywidgets import interact
```

76.2.1 Background

The model builds on work by Judd [93], Deneckner and Judd [44] and Helpman and Krugman [83] 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.

76.3 Key Ideas

It is helpful to begin with an overview of the mechanism.

76.3.1 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.

76.3.2 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.

76.4 Model

Let's write down the model more formally.

(The treatment is relatively terse since full details can be found in the original paper)

Time is discrete with $t = 0, 1, \ldots$.

There are two countries indexed by j or k.

In each country, a representative household inelastically supplies L_j units of labor at wage rate $w_{j,t}$.

Without loss of generality, it is assumed that $L_1 \geq L_2$.

Households consume a single nontradeable final good which is produced competitively.

Its production involves combining two types of tradeable intermediate inputs via

$$Y_{k,t} = C_{k,t} = \left(\frac{X^o_{k,t}}{1-\alpha}\right)^{1-\alpha} \left(\frac{X_{k,t}}{\alpha}\right)^{\alpha}$$

Here $X^o_{k,t}$ is a homogeneous input which can be produced from labor using a linear, one-for-one technology.

It is freely tradeable, competitively supplied, and homogeneous across countries.

By choosing the price of this good as numeraire and assuming both countries find it optimal to always produce the homogeneous good, we can set $w_{1,t} = w_{2,t} = 1$.

The good $X_{k,t}$ is a composite, built from many differentiated goods via

$$X_{k,t}^{1-\frac{1}{\sigma}} = \int_{\Omega_t} \left[x_{k,t}(\nu)\right]^{1-\frac{1}{\sigma}} d\nu$$

Here $x_{k,t}(\nu)$ is the total amount of a differentiated good $\nu \in \Omega_t$ that is produced.

The parameter $\sigma > 1$ is the direct partial elasticity of substitution between a pair of varieties and Ω_t is the set of varieties available in period t.

We can split the varieties into those which are supplied competitively and those supplied monopolistically; that is, $\Omega_t = \Omega_t^c + \Omega_t^m$.

76.4.1 Prices

Demand for differentiated inputs is

$$x_{k,t}(\nu) = \left(\frac{p_{k,t}(\nu)}{P_{k,t}}\right)^{-\sigma} \frac{\alpha L_k}{P_{k,t}}$$

Here

- $p_{k,t}(\nu)$ is the price of the variety ν and
- $P_{k,t}$ is the price index for differentiated inputs in k, defined by

$$\left[P_{k,t}\right]^{1-\sigma} = \int_{\Omega_t} [p_{k,t}(\nu)]^{1-\sigma} d\nu$$

The price of a variety also depends on the origin, j, and destination, k, of the goods because shipping varieties between countries incurs an iceberg trade cost $\tau_{j,k}$.

Thus the effective price in country k of a variety ν produced in country j becomes $p_{k,t}(\nu) = \tau_{j,k}\, p_{j,t}(\nu)$.

Using these expressions, we can derive the total demand for each variety, which is

$$D_{j,t}(\nu) = \sum_k \tau_{j,k} x_{k,t}(\nu) = \alpha A_{j,t}(p_{j,t}(\nu))^{-\sigma}$$

where

$$A_{j,t} := \sum_k \frac{\rho_{j,k} L_k}{(P_{k,t})^{1-\sigma}} \quad \text{and} \quad \rho_{j,k} = (\tau_{j,k})^{1-\sigma} \leq 1$$

It is assumed that $\tau_{1,1} = \tau_{2,2} = 1$ and $\tau_{1,2} = \tau_{2,1} = \tau$ for some $\tau > 1$, so that

$$\rho_{1,2} = \rho_{2,1} = \rho := \tau^{1-\sigma} < 1$$

The value $\rho \in [0, 1)$ is a proxy for the degree of globalization.

Producing one unit of each differentiated variety requires ψ units of labor, so the marginal cost is equal to ψ for $\nu \in \Omega_{j,t}$.

Additionally, all competitive varieties will have the same price (because of equal marginal cost), which means that, for all $\nu \in \Omega^c$,

$$p_{j,t}(\nu) = p^c_{j,t} := \psi \quad \text{and} \quad D_{j,t} = y^c_{j,t} := \alpha A_{j,t}(p^c_{j,t})^{-\sigma}$$

Monopolists will have the same marked-up price, so, for all $\nu \in \Omega^m$,

$$p_{j,t}(\nu) = p^m_{j,t} := \frac{\psi}{1 - \frac{1}{\sigma}} \quad \text{and} \quad D_{j,t} = y^m_{j,t} := \alpha A_{j,t}(p^m_{j,t})^{-\sigma}$$

Define

$$\theta := \frac{p^c_{j,t}}{p^m_{j,t}} \frac{y^c_{j,t}}{y^m_{j,t}} = \left(1 - \frac{1}{\sigma}\right)^{1-\sigma}$$

Using the preceding definitions and some algebra, the price indices can now be rewritten as

$$\left(\frac{P_{k,t}}{\psi}\right)^{1-\sigma} = M_{k,t} + \rho M_{j,t} \quad \text{where} \quad M_{j,t} := N^c_{j,t} + \frac{N^m_{j,t}}{\theta}$$

The symbols $N^c_{j,t}$ and $N^m_{j,t}$ will denote the measures of Ω^c and Ω^m respectively.

76.4.2 New Varieties

To introduce a new variety, a firm must hire f units of labor per variety in each country.

Monopolist profits must be less than or equal to zero in expectation, so

$$N^m_{j,t} \geq 0, \quad \pi^m_{j,t} := (p^m_{j,t} - \psi)y^m_{j,t} - f \leq 0 \quad \text{and} \quad \pi^m_{j,t}N^m_{j,t} = 0$$

With further manipulations, this becomes

$$N^m_{j,t} = \theta(M_{j,t} - N^c_{j,t}) \geq 0, \quad \frac{1}{\sigma}\left[\frac{\alpha L_j}{\theta(M_{j,t} + \rho M_{k,t})} + \frac{\alpha L_k}{\theta(M_{j,t} + M_{k,t}/\rho)}\right] \leq f$$

76.4.3 Law of Motion

With δ as the exogenous probability of a variety becoming obsolete, the dynamic equation for the measure of firms becomes

$$N^c_{j,t+1} = \delta(N^c_{j,t} + N^m_{j,t}) = \delta(N^c_{j,t} + \theta(M_{j,t} - N^c_{j,t}))$$

We will work with a normalized measure of varieties

$$n_{j,t} := \frac{\theta\sigma f N^c_{j,t}}{\alpha(L_1 + L_2)}, \quad i_{j,t} := \frac{\theta\sigma f N^m_{j,t}}{\alpha(L_1 + L_2)}, \quad m_{j,t} := \frac{\theta\sigma f M_{j,t}}{\alpha(L_1 + L_2)} = n_{j,t} + \frac{i_{j,t}}{\theta}$$

We also use $s_j := \frac{L_j}{L_1+L_2}$ to be the share of labor employed in country j.

We can use these definitions and the preceding expressions to obtain a law of motion for $n_t := (n_{1,t}, n_{2,t})$.

In particular, given an initial condition, $n_0 = (n_{1,0}, n_{2,0}) \in \mathbb{R}^2_+$, the equilibrium trajectory, $\{n_t\}_{t=0}^\infty = \{(n_{1,t}, n_{2,t})\}_{t=0}^\infty$, is obtained by iterating on $n_{t+1} = F(n_t)$ where $F : \mathbb{R}^2_+ \to \mathbb{R}^2_+$ is given by

$$F(n_t) = \begin{cases} (\delta(\theta s_1(\rho) + (1-\theta)n_{1,t}), \delta(\theta s_2(\rho) + (1-\theta)n_{2,t})) & \text{for } n_t \in D_{LL} \\ (\delta n_{1,t}, \delta n_{2,t}) & \text{for } n_t \in D_{HH} \\ (\delta n_{1,t}, \delta(\theta h_2(n_{1,t}) + (1-\theta)n_{2,t})) & \text{for } n_t \in D_{HL} \\ (\delta(\theta h_1(n_{2,t}) + (1-\theta)n_{1,t}, \delta n_{2,t})) & \text{for } n_t \in D_{LH} \end{cases}$$

Here

$$\begin{aligned} D_{LL} &:= \{(n_1, n_2) \in \mathbb{R}^2_+ | n_j \leq s_j(\rho)\} \\ D_{HH} &:= \{(n_1, n_2) \in \mathbb{R}^2_+ | n_j \geq h_j(\rho)\} \\ D_{HL} &:= \{(n_1, n_2) \in \mathbb{R}^2_+ | n_1 \geq s_1(\rho) \text{ and } n_2 \leq h_2(n_1)\} \\ D_{LH} &:= \{(n_1, n_2) \in \mathbb{R}^2_+ | n_1 \leq h_1(n_2) \text{ and } n_2 \geq s_2(\rho)\} \end{aligned}$$

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

$$h_j(n_k)^2 + \left((\rho + \frac{1}{\rho})n_k - s_j - s_k\right) h_j(n_k) + (n_k^2 - \frac{s_j n_k}{\rho} - s_k n_k \rho) = 0$$

76.5 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

In [2]:
```
@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))

@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
    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))
    # 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)
        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

        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
```

76.5.1 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

In [3]:
```
def plot_timeseries(n1_0, n2_0, s1=0.5, θ=2.5,
    δ=0.7, ρ=0.2, ax=None, title=''):
    """
    Plot a single time series with initial conditions
    """
    if ax is None:
        fig, ax = plt.subplots()

    # Create the MSG Model and simulate with initial conditions
    model = MSGSync(s1, θ, δ, ρ)
    n1, n2 = model.simulate_n(n1_0, n2_0, 25)

    ax.plot(np.arange(25), n1, label="$n_1$", lw=2)
    ax.plot(np.arange(25), n2, label="$n_2$", lw=2)

    ax.legend()
    ax.set(title=title, ylim=(0.15, 0.8))

    return ax


# Create figure
fig, ax = plt.subplots(2, 1, figsize=(10, 8))

plot_timeseries(0.15, 0.35, ax=ax[0], title='Not Synchronized')
plot_timeseries(0.4, 0.3, ax=ax[1], title='Synchronized')

fig.tight_layout()

plt.show()
```

In the first case, innovation in the two countries does not synchronize.

In the second case, different initial conditions are chosen, and the cycles become synchronized.

76.5.2 Basin of Attraction

Next, let's study the initial conditions that lead to synchronized cycles more systematically.

We generate time series from a large collection of different initial conditions and mark those conditions with different colors according to whether synchronization occurs or not.

The next display shows exactly this for four different parameterizations (one for each subfigure).

Dark colors indicate synchronization, while light colors indicate failure to synchronize.

As you can see, larger values of ρ translate to more synchronization.

You are asked to replicate this figure in the exercises.

In the solution to the exercises, you'll also find a figure with sliders, allowing you to experiment with different parameters.

Here's one snapshot from the interactive figure

76.6 Exercises

76.6.1 Exercise 1

Replicate the figure shown above by coloring initial conditions according to whether or not synchronization occurs from those conditions.

76.7 Solutions

These solutions are written by Chase Coleman.

```
In [4]: def plot_attraction_basis(s1=0.5, θ=2.5, δ=0.7, ρ=0.2, npts=250, ax=None):
            if ax is None:
                fig, ax = plt.subplots()

            # Create attraction basis
            unitrange = np.linspace(0, 1, npts)
            model = MSGSync(s1, θ, δ, ρ)
            ab = model.create_attraction_basis(npts=npts)
            cf = ax.pcolormesh(unitrange, unitrange, ab, cmap="viridis")

            return ab, cf


        fig = plt.figure(figsize=(14, 12))

        # Left - Bottom - Width - Height
        ax0 = fig.add_axes((0.05, 0.475, 0.38, 0.35), label="axes0")
        ax1 = fig.add_axes((0.5, 0.475, 0.38, 0.35), label="axes1")
```

```
ax2 = fig.add_axes((0.05, 0.05, 0.38, 0.35), label="axes2")
ax3 = fig.add_axes((0.5, 0.05, 0.38, 0.35), label="axes3")

params = [[0.5, 2.5, 0.7, 0.2],
          [0.5, 2.5, 0.7, 0.4],
          [0.5, 2.5, 0.7, 0.6],
          [0.5, 2.5, 0.7, 0.8]]

ab0, cf0 = plot_attraction_basis(*params[0], npts=500, ax=ax0)
ab1, cf1 = plot_attraction_basis(*params[1], npts=500, ax=ax1)
ab2, cf2 = plot_attraction_basis(*params[2], npts=500, ax=ax2)
ab3, cf3 = plot_attraction_basis(*params[3], npts=500, ax=ax3)

cbar_ax = fig.add_axes([0.9, 0.075, 0.03, 0.725])
plt.colorbar(cf0, cax=cbar_ax)

ax0.set_title(r"$s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.2$",
              fontsize=22)
ax1.set_title(r"$s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.4$",
              fontsize=22)
ax2.set_title(r"$s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.6$",
              fontsize=22)
ax3.set_title(r"$s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.8$",
              fontsize=22)

fig.suptitle("Synchronized versus Asynchronized 2-cycles",
             x=0.475, y=0.915, size=26)
plt.show()
```

76.7.1 Interactive Version

Additionally, instead of just seeing 4 plots at once, we might want to manually be able to change ρ and see how it affects the plot in real-time. Below we use an interactive plot to do this.

Note, interactive plotting requires the ipywidgets module to be installed and enabled.

```
In [5]: def interact_attraction_basis(ρ=0.2, maxiter=250, npts=250):
            # Create the figure and axis that we will plot on
            fig, ax = plt.subplots(figsize=(12, 10))

            # Create model and attraction basis
            s1, θ, δ = 0.5, 2.5, 0.75
            model = MSGSync(s1, θ, δ, ρ)
            ab = model.create_attraction_basis(maxiter=maxiter, npts=npts)

            # Color map with colormesh
            unitrange = np.linspace(0, 1, npts)
            cf = ax.pcolormesh(unitrange, unitrange, ab, cmap="viridis")
            cbar_ax = fig.add_axes([0.95, 0.15, 0.05, 0.7])
            plt.colorbar(cf, cax=cbar_ax)
            plt.show()
            return None
```

```
In [6]: fig = interact(interact_attraction_basis,
                       ρ=(0.0, 1.0, 0.05),
                       maxiter=(50, 5000, 50),
                       npts=(25, 750, 25))
```

Chapter 77

Coase's Theory of the Firm

77.1 Contents

- Overview 77.2
- The Model 77.3
- Equilibrium 77.4
- Existence, Uniqueness and Computation of Equilibria 77.5
- Implementation 77.6
- Exercises 77.7
- Solutions 77.8

77.2 Overview

In 1937, Ronald Coase wrote a brilliant essay on the nature of the firm [36].

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:

```
In [1]: import numpy as np
```

```
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.optimize import fminbound
from interpolation import interp
```

77.2.1 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.

77.2.2 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…" ([36], 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.

77.2.3 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.

77.2.4 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 [97].

77.3 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.

77.3.1 Subcontracting

The subcontracting scheme by which tasks are allocated across firms is illustrated in the figure below

In this example,

- Firm 1 receives a contract to sell one unit of the completed good to a final buyer.
- Firm 1 then forms a contract with firm 2 to purchase the partially completed good at stage t_1, with the intention of implementing the remaining $1 - t_1$ tasks in-house (i.e., processing from stage t_1 to stage 1).
- Firm 2 repeats this procedure, forming a contract with firm 3 to purchase the good at stage t_2.
- Firm 3 decides to complete the chain, selecting $t_3 = 0$.

At this point, production unfolds in the opposite direction (i.e., from upstream to downstream).

- Firm 3 completes processing stages from $t_3 = 0$ up to t_2 and transfers the good to firm 2.
- Firm 2 then processes from t_2 up to t_1 and transfers the good to firm 1,
- Firm 1 processes from t_1 to 1 and delivers the completed good to the final buyer.

The length of the interval of stages (range of tasks) carried out by firm i is denoted by ℓ_i.

Each firm chooses only its *upstream* boundary, treating its downstream boundary as given.

The benefit of this formulation is that it implies a recursive structure for the decision problem for each firm.

In choosing how many processing stages to subcontract, each successive firm faces essentially the same decision problem as the firm above it in the chain, with the only difference being

that the decision space is a subinterval of the decision space for the firm above.

We will exploit this recursive structure in our study of equilibrium.

77.3.2 Costs

Recall that we are considering a trade-off between two types of costs.

Let's discuss these costs and how we represent them mathematically.

Diminishing returns to management means rising costs per task when a firm expands the range of productive activities coordinated by its managers.

We represent these ideas by taking the cost of carrying out ℓ tasks in-house to be $c(\ell)$, where c is increasing and strictly convex.

Thus, the average cost per task rises with the range of tasks performed in-house.

We also assume that c is continuously differentiable, with $c(0) = 0$ and $c'(0) > 0$.

Transaction costs are represented as a wedge between the buyer's and seller's prices.

It matters little for us whether the transaction cost is borne by the buyer or the seller.

Here we assume that the cost is borne only by the buyer.

In particular, when two firms agree to a trade at face value v, the buyer's total outlay is δv, where $\delta > 1$.

The seller receives only v, and the difference is paid to agents outside the model.

77.4 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.

77.4.1 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

77.4.2 Formal Definition of Equilibrium

Let's make this definition more formal.

(You might like to skip this section on first reading)

An **allocation** of firms is a nonnegative sequence $\{\ell_i\}_{i\in\mathbb{N}}$ such that $\ell_i = 0$ for all sufficiently large i.

Recalling the figures above,

- ℓ_i represents the range of tasks implemented by the i-th firm

As a labeling convention, we assume that firms enter in order, with firm 1 being the furthest downstream.

An allocation $\{\ell_i\}$ is called **feasible** if $\sum_{i\geq 1} \ell_i = 1$.

In a feasible allocation, the entire production process is completed by finitely many firms.

Given a feasible allocation, $\{\ell_i\}$, let $\{t_i\}$ represent the corresponding transaction stages, defined by

$$t_0 = s \quad \text{and} \quad t_i = t_{i-1} - \ell_i \tag{1}$$

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

$$\pi_i = p(t_{i-1}) - c(\ell_i) - \delta p(t_i) \tag{2}$$

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 (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.

77.5 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?

77.5.1 A Fixed Point Method

To address these questions, we introduce the operator T mapping a nonnegative function p on $[0, 1]$ to Tp via

$$Tp(s) = \min_{t \leq s} \{c(s-t) + \delta p(t)\} \quad \text{for all} \quad s \in [0,1]. \tag{3}$$

Here and below, the restriction $0 \leq t$ in the minimum is understood.

The operator T is similar to a Bellman operator.

Under this analogy, p corresponds to a value function and δ to a discount factor.

But $\delta > 1$, so T is not a contraction in any obvious metric, and in fact, $T^n p$ diverges for many choices of p.

Nevertheless, there exists a domain on which T is well-behaved: the set of convex increasing continuous functions $p\colon [0,1] \to \mathbb{R}$ such that $c'(0)s \leq p(s) \leq c(s)$ for all $0 \leq s \leq 1$.

We denote this set of functions by $\mathcal{P}$.

In [97] 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

$$t^*(s) := \text{ the solution to } \min_{t \leq s} \{c(s-t) + \delta p^*(t)\} \tag{4}$$

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 (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

$$t_0^* = 1 \quad \text{and} \quad t_i^* = t^*(t_{i-1}) \tag{5}$$

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

$$n^* := \inf\{i \in \mathbb{N} \,:\, t_i^* = 0\} \tag{6}$$

The task allocation corresponding to (5) is given by $\ell_i^* := t_{i-1}^* - t_i^*$ for all i.

In [97] it is shown that

1. The value n^* in (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

$$p^*(s) = \min_{t \leq s} \{c(s-t) + \delta p^*(t)\} \quad \text{for all} \quad s \in [0,1] \tag{7}$$

From this equation, it is clear that so profits are zero for all incumbent firms.

77.5.2 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 [97] it is shown that t^* and ℓ^* are increasing and continuous, while p^* is continuously differentiable at all $s \in (0,1)$ with

$$(p^*)'(s) = c'(\ell^*(s)) \tag{8}$$

Equation (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

$$\delta (p^*)'(t^*(s)) = c'(s - t^*(s)) \tag{9}$$

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 (8) and (9) and evaluating at $s = t_i$, we see that active firms that are adjacent satisfy

$$\delta\, c'(\ell_{i+1}^*) = c'(\ell_i^*) \tag{10}$$

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.

77.6 Implementation

For most specifications of primitives, there is no closed-form solution for the equilibrium as far as we are aware.

However, we know that we can compute the equilibrium corresponding to a given transaction cost parameter δ and a cost function c by applying the results stated above.

In particular, we can

1. fix initial condition $p \in \mathcal{P}$,
2. iterate with T until $T^n p$ has converged to p^*, and
3. recover firm choices via the choice function (3)

As we step between iterates, we will use linear interpolation of functions, as we did in our lecture on optimal growth and several other places.

To begin, here's a class to store primitives and a grid

```
In [2]: class ProductionChain:

            def __init__(self,
                    n=1000,
                    delta=1.05,
                    c=lambda t: np.exp(10 * t) - 1):

                self.n, self.delta, self.c = n, delta, c
                self.grid = np.linspace(0, 1, n)
```

Now let's implement and iterate with T until convergence.

Recalling that our initial condition must lie in $\mathcal{P}$, we set $p_0 = c$

```
In [3]: 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 i, s in enumerate(grid):
            Tp = lambda t: delta * interp(grid, p, t) + c(s - t)
            new_p[i] = 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.

```
In [4]: def optimal_choices(pc, p_function, s):
            """
            Takes p_func as the true function, minimizes on [0,s]

            Returns optimal upstream boundary t_star and optimal size of
            firm ell_star

            In fact, the algorithm minimizes on [-1,s] and then takes the
            max of the minimizer and zero. This results in better results
            close to zero

            """
            delta, c = pc.delta, pc.c
            f = lambda t: delta * p_function(t) + c(s - t)
            t_star = max(fminbound(f, -1, s), 0)
            ell_star = s - t_star
            return t_star, ell_star
```

The allocation of firms can be computed by recursively stepping through firms' choices of their respective upstream boundary, treating the previous firm's upstream boundary as their own downstream boundary.

In doing so, we start with firm 1, who has downstream boundary $s = 1$.

```
In [5]: 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

In [6]:
```
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 1000 steps
```

Here's the function ℓ^*, which shows how large a firm with downstream boundary s chooses to be

In [7]:
```
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.

77.7 Exercises

77.7.1 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 δ increases? Why?

Check your intuition by computing the number of firms at delta in (1.01, 1.05, 1.1).

77.7.2 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.

77.8 Solutions

77.8.1 Exercise 1

```
In [8]: 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 1000 steps
When delta=1.01 there are 46 firms
Iteration converged in 1000 steps
When delta=1.05 there are 22 firms
Iteration converged in 1000 steps
When delta=1.1 there are 16 firms
```

77.8.2 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

```
In [9]: 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 1000 steps
```

2.5
2.0
1.5
1.0
0.5
0.0
downstream firms
upstream firms

Part X

Dynamic Linear Economies

Chapter 78

Recursive Models of Dynamic Linear Economies

78.1 Contents

- A Suite of Models 78.2
- Econometrics 78.3
- Dynamic Demand Curves and Canonical Household Technologies 78.4
- Gorman Aggregation and Engel Curves 78.5
- Partial Equilibrium 78.6
- Equilibrium Investment Under Uncertainty 78.7
- A Rosen-Topel Housing Model 78.8
- Cattle Cycles 78.9
- Models of Occupational Choice and Pay 78.10
- Permanent Income Models 78.11
- Gorman Heterogeneous Households 78.12
- Non-Gorman Heterogeneous Households 78.13

> "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

78.2 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 [78].

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

1. Lucas Asset Pricing using DLE
2. IRFs in Hall Model
3. Permanent Income Using the DLE class
4. Rosen schooling model
5. Cattle cycles
6. Shock Non Invertibility

78.2.1 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 [78] 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 [78] 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 [78] 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.

78.2.2 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.

78.2.3 Theory and Econometrics

For an application of the [78] 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.

78.2.4 More Details

A [78] 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.

[78] 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

78.2.5 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 $\mathcal{N}(Ax_t, CC')$.

78.2.6 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 $\{x_0, w_1, \dots, w_t\}$.

78.2.7 Prediction Theory

The optimal forecast of x_{t+1} given current information is

$$E(x_{t+1} \mid J_t) = Ax_t$$

and the one-step-ahead forecast error is

$$x_{t+1} - E(x_{t+1} \mid J_t) = Cw_{t+1}$$

The covariance matrix of x_{t+1} conditioned on J_t is

$$E(x_{t+1} - E(x_{t+1} \mid J_t))(x_{t+1} - E(x_{t+1} \mid J_t))' = CC'$$

A nonrecursive expression for x_t as a function of $x_0, w_1, w_2, \dots, w_t$ is

$$\begin{aligned} x_t &= Ax_{t-1} + Cw_t \\ &= A^2 x_{t-2} + ACw_{t-1} + Cw_t \\ &= \Big[\sum_{\tau=0}^{t-1} A^\tau C w_{t-\tau}\Big] + A^t x_0 \end{aligned}$$

Shift forward in time:

$$x_{t+j} = \sum_{s=0}^{j-1} A^s C w_{t+j-s} + A^j x_t$$

Projecting on the information set $\{x_0, w_t, w_{t-1}, \dots, w_1\}$ gives

$$E_t x_{t+j} = A^j x_t$$

where $E_t(\cdot) \equiv E[(\cdot) \mid x_0, w_t, w_{t-1}, \dots, w_1] = E(\cdot) \mid J_t$, and x_t is in J_t.

It is useful to obtain the covariance matrix of the j-step-ahead prediction error $x_{t+j} - E_t x_{t+j} = \sum_{s=0}^{j-1} A^s C w_{t-s+j}$.

Evidently,

$$E_t(x_{t+j} - E_t x_{t+j})(x_{t+j} - E_t x_{t+j})' = \sum_{k=0}^{j-1} A^k C C' A^{k'} \equiv v_j$$

v_j can be calculated recursively via

$$\begin{aligned} v_1 &= CC' \\ v_j &= CC' + A v_{j-1} A', \quad j \geq 2 \end{aligned}$$

78.2.8 Orthogonal Decomposition

To decompose these covariances into parts attributable to the individual components of w_t, we let i_τ be an N-dimensional column vector of zeroes except in position τ, where there is a one. Define a matrix $v_{j,\tau}$

$$v_{j,\tau} = \sum_{k=0}^{j-1} A^k C i_\tau i_\tau' C' A'^k.$$

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 $\tau = 1, \dots, N$.

78.2.9 Taste and Technology Shocks

$E(w_t \mid J_{t-1}) = 0$ and $E(w_t w_t' \mid J_{t-1}) = I$ for $t = 1, 2, ...$

$$b_t = U_b z_t \text{ and } d_t = U_d z_t,$$

U_b and U_d are matrices that select entries of z_t. The law of motion for $\{z_t : t = 0, 1, ...\}$ is

$$z_{t+1} = A_{22} z_t + C_2 w_{t+1} \quad \text{for } t = 0, 1, ...$$

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

78.2.10 Production Technology

Where c_t is a vector of consumption rates, k_t is a vector of physical capital goods, g_t is a vector intermediate productions goods, d_t is a vector of technology shocks, the production technology is

$$\begin{aligned} \Phi_c c_t + \Phi_g g_t + \Phi_i i_t &= \Gamma k_{t-1} + d_t \\ k_t &= \Delta_k k_{t-1} + \Theta_k i_t \\ g_t \cdot g_t &= \ell_t^2 \end{aligned}$$

Here $\Phi_c, \Phi_g, \Phi_i, \Gamma, \Delta_k, \Theta_k$ are all matrices conformable to the vectors they multiply and ℓ_t is a disutility generating resource supplied by the household.

For technical reasons that facilitate computations, we make the following.

Assumption: $[\Phi_c \ \Phi_g]$ is nonsingular.

78.2.11 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 Δ_h are less than or equal to one.

Below, we'll outline further assumptions that we shall occasionally impose.

78.2.12 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

$$\left(\frac{1}{2}\right) E \sum_{t=0}^{\infty} \beta^t [(s_t - b_t) \cdot (s_t - b_t) + \ell_t^2] \big| J_0, \ 0 < \beta < 1$$

We now proceed to give examples of production and household technologies that appear in various models that appear in the literature.

First, we give examples of production Technologies

$$\Phi_c c_t + \Phi_g g_t + \Phi_i i_t = \Gamma k_{t-1} + d_t$$

$$\mid g_t \mid \leq \ell_t$$

so we'll be looking for specifications of the matrices $\Phi_c, \Phi_g, \Phi_i, \Gamma, \Delta_k, \Theta_k$ that define them.

78.2.13 Endowment Economy

There is a single consumption good that cannot be stored over time.

In time period t, there is an endowment d_t of this single good.

There is neither a capital stock, nor an intermediate good, nor a rate of investment.

So $c_t = d_t$.

To implement this specification, we can choose A_{22}, C_2, and U_d to make d_t follow any of a variety of stochastic processes.

To satisfy our earlier rank assumption, we set:

$$c_t + i_t = d_{1t}$$

$$g_t = \phi_1 i_t$$

where ϕ_1 is a small positive number.

To implement this version, we set $\Delta_k = \Theta_k = 0$ and

$$\Phi_c = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \Phi_i = \begin{bmatrix} 1 \\ \phi_1 \end{bmatrix}, \quad \Phi_g = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad \Gamma = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad d_t = \begin{bmatrix} d_{1t} \\ 0 \end{bmatrix}$$

We can use this specification to create a linear-quadratic version of Lucas's (1978) asset pricing model.

78.2.14 Single-Period Adjustment Costs

There is a single consumption good, a single intermediate good, and a single investment good.

The technology is described by

$$\begin{aligned} c_t &= \gamma k_{t-1} + d_{1t}, \quad \gamma > 0 \\ \phi_1 i_t &= g_t + d_{2t}, \quad \phi_1 > 0 \\ \ell_t^2 &= g_t^2 \\ k_t &= \delta_k k_{t-1} + i_t, \ 0 < \delta_k < 1 \end{aligned}$$

Set

$$\Phi_c = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \ \Phi_g = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \ \Phi_i = \begin{bmatrix} 0 \\ \phi_1 \end{bmatrix}$$

$$\Gamma = \begin{bmatrix} \gamma \\ 0 \end{bmatrix}, \ \Delta_k = \delta_k, \ \Theta_k = 1$$

We set A_{22}, C_2 and U_d to make $(d_{1t}, d_{2t})' = d_t$ follow a desired stochastic process.

Now we describe some examples of preferences, which as we have seen are ordered by

$$-\left(\frac{1}{2}\right) E \sum_{t=0}^{\infty} \beta^t \left[(s_t - b_t) \cdot (s_t - b_t) + (\ell_t)^2\right] \mid J_0 \quad , \ 0 < \beta < 1$$

where household services are produced via the household technology

$$h_t = \Delta_h h_{t-1} + \Theta_h c_t$$

$$s_t = \Lambda h_{t-1} + \Pi c_t$$

and we make

Assumption: The absolute values of the eigenvalues of Δ_h are less than or equal to one.

Later we shall introduce **canonical** household technologies that satisfy an 'invertibility' requirement relating sequences $\{s_t\}$ of services and $\{c_t\}$ of consumption flows.

And we'll describe how to obtain a canonical representation of a household technology from one that is not canonical.

Here are some examples of household preferences.

Time Separable preferences

$$-\frac{1}{2} E \sum_{t=0}^{\infty} \beta^t \left[(c_t - b_t)^2 + \ell_t^2\right] \mid J_0 \quad , \ 0 < \beta < 1$$

Consumer Durables

$$h_t = \delta_h h_{t-1} + c_t \quad , \ 0 < \delta_h < 1$$

Services at t are related to the stock of durables at the beginning of the period:

$$s_t = \lambda h_{t-1} \ , \ \lambda > 0$$

Preferences are ordered by

$$-\frac{1}{2} E \sum_{t=0}^{\infty} \beta^t \left[(\lambda h_{t-1} - b_t)^2 + \ell_t^2 \right] \mid J_0$$

Set $\Delta_h = \delta_h, \Theta_h = 1, \Lambda = \lambda, \Pi = 0$.

Habit Persistence

$$-\Big(\frac{1}{2}\Big) E \sum_{t=0}^{\infty} \beta^t \Big[\big(c_t - \lambda(1-\delta_h) \sum_{j=0}^{\infty} \delta_h^j \, c_{t-j-1} - b_t\big)^2 + \ell_t^2 \Big] \big| J_0$$

$$0 < \beta < 1 \ , \ 0 < \delta_h < 1 \ , \ \lambda > 0$$

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

$$\Lambda = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \text{and} \quad \Pi = \begin{bmatrix} \pi_1 & 0 \\ \pi_2 & \pi_3 \end{bmatrix}$$

$$-\frac{1}{2}\beta^t(\Pi c_t - b_t)'(\Pi c_t - b_t)$$

$$mu_t = -\beta^t[\Pi'\Pi\, c_t - \Pi'\, b_t]$$

$$c_t = -(\Pi'\Pi)^{-1}\beta^{-t} mu_t + (\Pi'\Pi)^{-1}\Pi' b_t$$

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 π_2 and π_3 have the same sign, the goods are substitutes.

If they have opposite signs, the goods are complements.

To summarize, our economic structure consists of the matrices that define the following components:

Information and shocks

$$\begin{aligned} z_{t+1} &= A_{22} z_t + C_2 w_{t+1} \\ b_t &= U_b z_t \\ d_t &= U_d z_t \end{aligned}$$

Production Technology

$$\begin{aligned} \Phi_c c_t + \Phi_g g_t + \Phi_i i_t &= \Gamma k_{t-1} + d_t \\ k_t &= \Delta_k k_{t-1} + \Theta_k i_t \\ g_t \cdot g_t &= \ell_t^2 \end{aligned}$$

Household Technology

$$\begin{aligned} s_t &= \Lambda h_{t-1} + \Pi c_t \\ h_t &= \Delta_h h_{t-1} + \Theta_h c_t \end{aligned}$$

Preferences

$$\left(\frac{1}{2}\right)E\sum_{t=0}^{\infty}\beta^t[(s_t-b_t)\cdot(s_t-b_t)+\ell_t^2]\big|J_0,\ 0<\beta<1$$

*Next steps:** we move on to discuss two closely connected concepts

- A Planning Problem or Optimal Resource Allocation Problem
- Competitive Equilibrium

78.2.15 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

78.2.16 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**

$$
\begin{aligned}
-\Phi_c' M_t^d + \Theta_h' M_t^h + \Pi' M_t^s &= 0,\\
-g_t - \Phi_g' M_t^d &= 0,\\
-M_t^h + \beta E(\Delta_h' M_{t+1}^h + \Lambda' M_{t+1}^s) \mid J_t &= 0,\\
-\Phi_i' M_t^d + \Theta_k' M_t^k &= 0,\\
-M_t^k + \beta E(\Delta_k' M_{t+1}^k + \Gamma' M_{t+1}^d) \mid J_t &= 0,\\
-s_t + b_t - M_t^s &= 0\\
\Phi_c c_t + \Phi_g g_t + \Phi_i i_t &= \Gamma k_{t-1} + d_t,\\
k_t &= \Delta_k k_{t-1} + \Theta_k i_t,\\
h_t &= \Delta_h h_{t-1} + \Theta_h c_t,\\
s_t &= \Lambda h_{t-1} + \Pi c_t,\\
z_{t+1} &= A_{22} z_t + C_2 w_{t+1},\ b_t = U_b z_t, \quad \text{and} \quad d_t = U_d z_t
\end{aligned}
$$

The Lagrange multipliers or **shadow prices** satisfy

$$M_t^s = b_t - s_t$$

$$M_t^h = E\left[\sum_{\tau=1}^{\infty} \beta^\tau (\Delta_h')^{\tau-1} \Lambda' M_{t+\tau}^s \mid J_t\right]$$

$$M_t^d = \begin{bmatrix} \Phi_c' \\ \Phi_g' \end{bmatrix}^{-1} \begin{bmatrix} \Theta_h' M_t^h + \Pi' M_t^s \\ -g_t \end{bmatrix}$$

$$M_t^k = E\left[\sum_{\tau=1}^{\infty} \beta^\tau (\Delta_k')^{\tau-1} \Gamma' M_{t+\tau}^d \mid J_t\right]$$

$$M_t^i = \Theta_k' M_t^k$$

Although it is possible to use matrix operator methods to solve the above **Python ready equations**, that is not the approach we'll use.

Instead, we'll use dynamic programming to get recursive representations for both quantities and shadow prices.

78.2.17 Dynamic Programming

Dynamic Programming always starts with the word **let**.

Thus, let $V(x_0)$ be the optimal value function for the planning problem as a function of the initial state vector x_0.

(Thus, in essence, dynamic programming amounts to an application of a **guess and verify** method in which we begin with a guess about the answer to the problem we want to solve. That's why we start with **let** $V(x_0)$ be the (value of the) answer to the problem, then establish and verify a bunch of conditions $V(x_0)$ has to satisfy if indeed it is the answer)

The optimal value function $V(x)$ satisfies the **Bellman equation**

$$V(x_0) = \max_{c_0, i_0, g_0} [-.5[(s_0 - b_0) \cdot (s_0 - b_0) + g_0 \cdot g_0] + \beta E V(x_1)]$$

subject to the linear constraints

$$\begin{aligned} \Phi_c c_0 + \Phi_g g_0 + \Phi_i i_0 &= \Gamma k_{-1} + d_0, \\ k_0 &= \Delta_k k_{-1} + \Theta_k i_0, \\ h_0 &= \Delta_h h_{-1} + \Theta_h c_0, \\ s_0 &= \Lambda h_{-1} + \Pi c_0, \\ z_1 &= A_{22} z_0 + C_2 w_1, \ b_0 = U_b z_0 \ \text{ and } \ d_0 = U_d z_0 \end{aligned}$$

Because this is a linear-quadratic dynamic programming problem, it turns out that the value function has the form

$$V(x) = x' P x + \rho$$

Thus, we want to solve an instance of the following linear-quadratic dynamic programming problem:

Choose a contingency plan for $\{x_{t+1}, u_t\}_{t=0}^{\infty}$ to maximize

$$-E\sum_{t=0}^{\infty}\beta^t[x_t'Rx_t + u_t'Qu_t + 2u_t'W'x_t],\ 0<\beta<1$$

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 ρ_j satisfy the equations

$$
\begin{aligned}
P_{j+1} &= R + \beta A' P_j A - (\beta A' P_j B + W)(Q + \beta B' P_j B)^{-1}(\beta B' P_j A + W') \\
\rho_{j+1} &= \beta \rho_j + \beta \text{ trace } P_j C C'
\end{aligned}
$$

We can now state the planning problem as a dynamic programming problem

$$
\max_{\{u_t, x_{t+1}\}} \; -E \sum_{t=0}^{\infty} \beta^t [x_t' R x_t + u_t' Q u_t + 2 u_t' W' x_t], \quad 0 < \beta < 1
$$

where maximization is subject to

$$
x_{t+1} = A x_t + B u_t + C w_{t+1}, \; t \geq 0
$$

$$
x_t = \begin{bmatrix} h_{t-1} \\ k_{t-1} \\ z_t \end{bmatrix}, \qquad u_t = i_t
$$

where

$$
\begin{aligned}
A &= \begin{bmatrix} \Delta_h & \Theta_h U_c [\Phi_c \;\; \Phi_g]^{-1} \Gamma & \Theta_h U_c [\Phi_c \;\; \Phi_g]^{-1} U_d \\ 0 & \Delta_k & 0 \\ 0 & 0 & A_{22} \end{bmatrix} \\
B &= \begin{bmatrix} -\Theta_h U_c [\Phi_c \;\; \Phi_g]^{-1} \Phi_i \\ \Theta_k \\ 0 \end{bmatrix}, \; C = \begin{bmatrix} 0 \\ 0 \\ C_2 \end{bmatrix}
\end{aligned}
$$

$$
\begin{bmatrix} x_t \\ u_t \end{bmatrix}' S \begin{bmatrix} x_t \\ u_t \end{bmatrix} = \begin{bmatrix} x_t \\ u_t \end{bmatrix}' \begin{bmatrix} R & W \\ W' & Q \end{bmatrix} \begin{bmatrix} x_t \\ u_t \end{bmatrix}
$$

$$
S = (G'G + H'H)/2
$$

$$
H = [\Lambda \;\vdots\; \Pi U_c [\Phi_c \;\; \Phi_g]^{-1} \Gamma \;\vdots\; \Pi U_c [\Phi_c \;\; \Phi_g]^{-1} U_d - U_b \;\vdots\; -\Pi U_c [\Phi_c \;\; \Phi_g]^{-1} \Phi_i]
$$

$$
G = U_g [\Phi_c \;\; \Phi_g]^{-1} [0 \;\vdots\; \Gamma \;\vdots\; U_d \;\vdots\; -\Phi_i].
$$

Lagrange multipliers as gradient of value function

A useful fact is that Lagrange multipliers equal gradients of the planner's value function

$$
\begin{aligned}
&\mathcal{M}_t^k = M_k x_t \;\text{ and }\; M_t^h = M_h x_t \;\text{ where} \\
&M_k = 2\beta [0 \; I \; 0] P A^o \\
&M_h = 2\beta [I \; 0 \; 0] P A^o
\end{aligned}
$$

$$
\mathcal{M}_t^s = M_s x_t \;\text{ where }\; M_s = (S_b - S_s) \;\text{ and }\; S_b = [0 \; 0 \; U_b]
$$

$$\mathcal{M}_t^d = M_d x_t \quad \text{where} \quad M_d = \begin{bmatrix} \Phi_c' \\ \Phi_g' \end{bmatrix}^{-1} \begin{bmatrix} \Theta_h' M_h + \Pi' M_s \\ -S_g \end{bmatrix}$$

$$\mathcal{M}_t^c = M_c x_t \quad \text{where} \quad M_c = \Theta_h' M_h + \Pi' M_s$$

$$\mathcal{M}_t^i = M_i x_t \quad \text{where} \quad M_i = \Theta_k' M_k$$

We will use this fact and these equations to compute competitive equilibrium prices.

78.2.18 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**.

78.2.19 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.

78.2.20 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

$$\Phi_c \, c_t + \Phi_g \, g_t + \Phi_i \, i_t = \Gamma k_{t-1} + d_t$$

$$-\ell_t^2 + g_t \cdot g_t = 0$$

78.2.21 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

$$k_t = \Delta_k k_{t-1} + \Theta_k i_t$$

78.2.22 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.

78.2.23 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:

$$a_0 = x_0' \, \mu_a \, x_0 + \sigma_a$$

$$\mu_a = \sum_{\tau=0}^{\infty} \beta^\tau \, (A^{o\prime})^\tau \; Z_a \, A^{o\tau}$$

$$\sigma_a = \frac{\beta}{1-\beta} \text{ trace} \left(Z_a \sum_{\tau=0}^{\infty} \beta^\tau \, (A^o)^\tau \, CC' (A^{o\prime})^\tau \right)$$

78.2.24 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

$$\begin{aligned} L_t^2 = [\{y_s\}_{s=t}^{\infty} : \; & y_s \;\text{ is a random variable in }\; J_s \;\text{ for }\; s \geq t \\ \text{and } E \sum_{s=t}^{\infty} \beta^{s-t} \; & y_s^2 \mid J_t < +\infty]. \end{aligned}$$

$$p_s^t = M_c x_s / [\bar{e}_j M_c x_t], \qquad s \geq t$$

$$w_s^t = \mid S_g x_s \mid / [\bar{e}_j M_c x_t], \quad s \geq t$$

$$r_s^t = \Gamma' M_d x_s / [\bar{e}_j M_c x_t], \quad s \geq t$$

$$q_s^t = M_i x_s / [\bar{e}_j \, M_c x_t], \qquad s \geq t$$

$$\alpha_s^t = M_d x_s / [\bar{e}_j \, M_c x_t], \quad s \geq t$$

$$v_t = [\Gamma' M_d + \Delta_k' M_k] x_t / \, [\bar{e}_j \, M_c x_t]$$

78.3 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 [78] 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 $a_t = y_t - E[y_t | y^{t-1}], E a_t a_t' \equiv \Omega_t = G \Sigma_t G' + R$.

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:

$$\begin{aligned} \Sigma_{t+1} &= A^o \Sigma_t A^{o\prime} + C C' \\ &\quad - A^o \Sigma_t G' (G \Sigma_t G' + R)^{-1} G \Sigma_t A^{o\prime} \end{aligned}$$

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

$$\begin{aligned}\Sigma &= A^o \Sigma A^{o\prime} + CC' \\ &\quad - A^o \Sigma G'(G\Sigma G' + R)^{-1} G\Sigma A^{o\prime} \\ K &= A^o \Sigma_t G'(G\Sigma G' + R)^{-1}\end{aligned}$$

$$\begin{aligned}\hat{x}_{t+1} &= A^o \hat{x}_t + K a_t \\ y_t &= G\hat{x}_t + a_t\end{aligned}$$

where $Ea_t a_t' \equiv \Omega = G\Sigma G' + R$.

78.3.1 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:

$$-.5 \sum_{t=0}^{T} \left\{ n_y \ln(2\pi) + \ln|\Omega_t| + a_t' \Omega_t^{-1} a_t \right\}$$

78.3.2 Covariance Generating Functions

Autocovariance: $C_x(\tau) = Ex_t x_{t-\tau}'$.
Generating Function: $S_x(z) = \sum_{\tau=-\infty}^{\infty} C_x(\tau) z^\tau, z \in C$.

78.3.3 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

78.3.4 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**

$$y_t = \sum_{j=1}^{\infty} G(A^o - KG)^{j-1} K y_{t-j} + a_t$$

78.4 Dynamic Demand Curves and Canonical Household Technologies

78.4.1 Canonical Household Technologies

$$\begin{aligned} h_t &= \Delta_h h_{t-1} + \Theta_h c_t \\ s_t &= \Lambda h_{t-1} + \Pi c_t \\ b_t &= U_b z_t \end{aligned}$$

Definition: A household service technology $(\Delta_h, \Theta_h, \Pi, \Lambda, U_b)$ is said to be **canonical** if

- Π is nonsingular, and
- the absolute values of the eigenvalues of $(\Delta_h - \Theta_h \Pi^{-1} \Lambda)$ are strictly less than $1/\sqrt{\beta}$.

Key invertibility property: A canonical household service technology maps a service process $\{s_t\}$ in L_0^2 into a corresponding consumption process $\{c_t\}$ for which the implied household capital stock process $\{h_t\}$ is also in L_0^2.

An inverse household technology:

$$
\begin{aligned}
c_t &= -\Pi^{-1}\Lambda h_{t-1} + \Pi^{-1}s_t \\
h_t &= (\Delta_h - \Theta_h\Pi^{-1}\Lambda)h_{t-1} + \Theta_h\Pi^{-1}s_t
\end{aligned}
$$

The restriction on the eigenvalues of the matrix $(\Delta_h - \Theta_h\Pi^{-1}\Lambda)$ keeps the household capital stock $\{h_t\}$ in L_0^2.

78.4.2 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 $h_{i,-1} = h_{-1}$.

$$
W_0 = E_0 \sum_{t=0}^{\infty} \beta^t (w_t^0 \ell_t + \alpha_t^0 \cdot d_t) + v_0 \cdot k_{-1}
$$

$$
\mu_0^w = \frac{E_0 \sum_{t=0}^{\infty} \beta^t \rho_t^0 \cdot (b_t - s_{i,t}) - W_0}{E_0 \sum_{t=0}^{\infty} \beta^t \rho_t^0 \cdot \rho_t^0}
$$

$$
\begin{aligned}
c_t = {} & -\Pi^{-1}\Lambda h_{t-1} + \Pi^{-1}b_t - \Pi^{-1}\mu_0^w E_t\{\Pi'^{-1} - \Pi'^{-1}\Theta_h' \\
& [I - (\Delta_h' - \Lambda'\Pi'^{-1}\Theta_h')\beta L^{-1}]^{-1}\Lambda'\Pi'^{-1}\beta L^{-1}\}p_t^0 \\
h_t = {} & \Delta_h h_{t-1} + \Theta_h c_t
\end{aligned}
$$

This system expresses consumption demands at date t as functions of: (i) time-t conditional expectations of future scaled Arrow-Debreu prices $\{p_{t+s}^0\}_{s=0}^{\infty}$; (ii) the stochastic process for the household's endowment $\{d_t\}$ and preference shock $\{b_t\}$, as mediated through the multiplier μ_0^w and wealth W_0; and (iii) past values of consumption, as mediated through the state variable h_{t-1}.

78.5 Gorman Aggregation and Engel Curves

We shall explore how the dynamic demand schedule for consumption goods opens up the possibility of satisfying Gorman's (1953) conditions for aggregation in a heterogeneous consumer model.

The first equation of our demand system is an Engel curve for consumption that is linear in the marginal utility μ_0^2 of individual wealth with a coefficient on μ_0^w that depends only on prices.

The multiplier μ_0^w depends on wealth in an affine relationship, so that consumption is linear in wealth.

In a model with multiple consumers who have the same household technologies $(\Delta_h, \Theta_h, \Lambda, \Pi)$ but possibly different preference shock processes and initial values of household capital stocks, the coefficient on the marginal utility of wealth is the same for all consumers.

Gorman showed that when Engel curves satisfy this property, there exists a unique community or aggregate preference ordering over aggregate consumption that is independent of the distribution of wealth.

78.5.1 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

$$W_t = E_t \sum_{j=0}^{\infty} \beta^j (w_{t+j}^t \ell_{t+j} + \alpha_{t+j}^t \cdot d_{t+j}) + v_t \cdot k_{t-1}$$

$$\mu_t^w = \frac{E_t \sum_{j=0}^{\infty} \beta^j \rho_{t+j}^t \cdot (b_{t+j} - s_{i,t+j}) - W_t}{E_t \sum_{t=0}^{\infty} \beta^j \rho_{t+j}^t \cdot \rho_{t+j}^t}$$

$$\begin{aligned} c_t &= -\Pi^{-1}\Lambda h_{t-1} + \Pi^{-1} b_t - \Pi^{-1}\mu_t^w E_t\{\Pi'^{-1} - \Pi'^{-1}\Theta_h' \\ &\quad [I - (\Delta_h' - \Lambda'\Pi'^{-1}\Theta_h')\beta L^{-1}]^{-1}\Lambda'\Pi'^{-1}\beta L^{-1}\} p_t^t \\ h_t &= \Delta_h h_{t-1} + \Theta_h c_t \end{aligned}$$

78.5.2 Dynamic Demand

Define a time t continuation of a sequence $\{z_t\}_{t=0}^{\infty}$ as the sequence $\{z_\tau\}_{\tau=t}^{\infty}$. The demand system indicates that the time t vector of demands for c_t is influenced by:

Through the multiplier μ_t^w, the time t continuation of the preference shock process $\{b_t\}$ and the time t continuation of $\{s_{i,t}\}$.

The time $t-1$ level of household durables h_{t-1}.

Everything that affects the household's time t wealth, including its stock of physical capital k_{t-1} and its value v_t, the time t continuation of the factor prices $\{w_t, \alpha_t\}$, the household's continuation endowment process, and the household's continuation plan for $\{\ell_t\}$.

The time t continuation of the vector of prices $\{p_t^t\}$.

78.5.3 Attaining a Canonical Household Technology

Apply the following version of a factorization identity:

$$\begin{aligned} &'[\Pi + \beta^{1/2} L\Lambda(I - \beta^{1/2} L\Delta_h)^{-1}\Theta_h] \\ &\qquad = [\hat{\Pi} + \beta^{1/2} L^{-1}\hat{\Lambda}(I - \beta^{1/2} L^{-1}\Delta_h)^{-1}\Theta_h]'[\hat{\Pi} + \beta^{1/2} L\hat{\Lambda}(I - \beta^{1/2} L\Delta_h)^{-1}\Theta_h] \end{aligned}$$

The factorization identity guarantees that the $[\hat{\Lambda}, \hat{\Pi}]$ representation satisfies both requirements for a canonical representation.

78.6 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

1. Lucas Asset Pricing using DLE
2. IRFs in Hall Model
3. Permanent Income Using the DLE class
4. Rosen schooling model
5. Cattle cycles
6. 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

$$\begin{aligned} \Phi_c c_t + \Phi_i i_t + \Phi_g g_t &= \Gamma k_{t-1} + d_t \\ k_t &= \Delta_k k_{t-1} + \Theta_k i_t. \end{aligned}$$

78.7 Equilibrium Investment Under Uncertainty

A representative firm maximizes

$$E\sum_{t=0}^{\infty}\beta^t\{p_tc_t - g_t^2/2\}$$

subject to the technology

$$\begin{aligned} c_t &= \gamma k_{t-1} \\ k_t &= \delta_k k_{t-1} + i_t \\ g_t &= f_1 i_t + f_2 d_t \end{aligned}$$

where d_t is a cost shifter, $\gamma > 0$, and $f_1 > 0$ is a cost parameter and $f_2 = 1$. Demand is governed by

$$p_t = \alpha_0 - \alpha_1 c_t + u_t$$

where u_t is a demand shifter with mean zero and α_0, α_1 are positive parameters.

Assume that u_t, d_t are uncorrelated first-order autoregressive processes.

78.8 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 δ_h is a depreciation factor for houses.

We cast this demand specification within our class of models by letting the stock of houses h_t evolve according to

$$h_t = \delta_h h_{t-1} + c_t, \quad \delta_h \in (0,1)$$

where c_t is the rate of production of new houses.

Houses produce services s_t according to $s_t = \bar{\lambda}h_t$ or $s_t = \lambda h_{t-1} + \pi c_t$, where $\lambda = \bar{\lambda}\delta_h, \pi = \bar{\lambda}$.

We can take $\bar{\lambda}\rho_t^0 = R_t$ as the rental rate on housing at time t, measured in units of time t consumption (housing).

Demand for housing services is

$$s_t = b_t - \mu_0\rho_t^0$$

where the price of new houses p_t is related to ρ_t^0 by $\rho_t^0 = \pi^{-1}[p_t - \beta\delta_h E_t p_{t+1}]$.

78.9 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

$$\begin{aligned} E_0 \sum_{t=0}^{\infty} \beta^t \{ & p_t c_t - \tilde{h}_t \tilde{x}_t - (\gamma_0 \tilde{h}_t)(g\tilde{x}_{t-1}) - (\gamma_1 \tilde{h}_t)(g\tilde{x}_{t-2}) - m_t c_t \\ & - \Psi(\tilde{x}_t, \tilde{x}_{t-1}, \tilde{x}_{t-2}, c_t)\} \end{aligned}$$

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

78.10 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

78.10.1 Market for Engineers

Ryoo and Rosen's (2004) [138] 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 $N_{-1}, n_{-s}, s = 1, \dots, -k$.

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.

$$s_t = [\lambda_1 \ 0 \ \dots \ 0] \begin{bmatrix} h_{1t-1} \\ h_{2t-1} \\ \vdots \\ h_{k+1,t-1} \end{bmatrix} + 0 \cdot c_t$$

$$\begin{bmatrix} h_{1t} \\ h_{2t} \\ \vdots \\ h_{k,t} \\ h_{k+1,t} \end{bmatrix} = \begin{bmatrix} \delta_N & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & 1 \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} h_{1t-1} \\ h_{2t-1} \\ \vdots \\ h_{k,t-1} \\ h_{k+1,t-1} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} c_t$$

This specification sets Rosen's $N_t = h_{1t-1}, n_t = c_t, h_{\tau+1,t-1} = n_{t-\tau}, \tau = 1, \dots, k$, and uses the home-produced service to capture the demand for labor. Here λ_1 embodies Rosen's demand parameter α_d.

- The supply of new workers becomes our consumption.
- The dynamic demand curve becomes Rosen's dynamic supply curve for new workers.

Remark: This has an Imai-Keane flavor.

For more details and Python code see Rosen schooling model.

78.10.2 Skilled and Unskilled Workers

First, a demand curve for labor

$$\begin{bmatrix} w_{ut} \\ w_{st} \end{bmatrix} = \alpha_d \begin{bmatrix} N_{ut} \\ N_{st} \end{bmatrix} + \epsilon_{1t}$$

where α_d is a (2×2) matrix of demand parameters and ϵ_{1t} is a vector of demand shifters second, time-to-train specifications for skilled and unskilled labor, respectively:

$$\begin{aligned} N_{st+k} &= \delta_N N_{st+k-1} + n_{st} \\ N_{ut} &= \delta_N N_{ut-1} + n_{ut}; \end{aligned}$$

where N_{st}, N_{ut} are stocks of the two types of labor, and n_{st}, n_{ut} are entry rates into the two occupations.

third, definitions of discounted present values of new entrants to the skilled and unskilled occupations, respectively:

$$\begin{aligned} v_{st} &= E_t \beta^k \sum_{j=0}^{\infty} (\beta \delta_N)^j w_{st+k+j} \\ v_{ut} &= E_t \sum_{j=0}^{\infty} (\beta \delta_N)^j w_{ut+j} \end{aligned}$$

where w_{ut}, w_{st} are wage rates for the two occupations; and fourth, supply curves for new entrants:

$$\begin{bmatrix} n_{st} \\ n_{ut} \end{bmatrix} = \alpha_s \begin{bmatrix} v_{ut} \\ v_{st} \end{bmatrix} + \epsilon_{2t}$$

Short Cut

As an alternative, Siow simply used the **equalizing differences** condition

$$v_{ut} = v_{st}$$

78.11 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

$$\mathcal{M}_t^k = E(\mathcal{M}_{t+1}^k | J_t)$$
$$\mathcal{M}_t^e = E(\mathcal{M}_{t+1}^e | J_t)$$

and

$$\mathcal{M}_t^c = (\Phi_c)' \mathcal{M}_t^d = \phi_c M_t^e$$

For more details see Permanent Income Using the DLE class

Testing Permanent Income Models:

We have two types of implications of permanent income models:

- Equality of present values of moving average coefficients.
- Martingale $\mathcal{M}_t^k$.

These have been tested in work by Hansen, Sargent, and Roberts (1991) [140] and by Attanasio and Pavoni (2011) [12].

78.12 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

$$b_{jt} = U_{bj} z_t$$

$$E \sum_{t=0}^{\infty} \beta^t \, p_t^0 \, \cdot c_{jt} \mid J_0 = E \sum_{t=0}^{\infty} \beta^t \, (w_t^0 \, \ell_{jt} + \alpha_t^0 \, \cdot d_{jt}) \mid J_0 + v_0 \, \cdot k_{j,-1},$$

where $k_{j,-1}$ is given. The jth consumer owns an endowment process d_{jt}, governed by the stochastic process $d_{jt} = U_{dj} \, z_t$.

We refer to this as a setting with Gorman heterogeneous households.

This specification confines heterogeneity among consumers to:

- differences in the preference processes $\{b_{jt}\}$, represented by different selections of U_{bj}
- differences in the endowment processes $\{d_{jt}\}$, represented by different selections of U_{dj}
- differences in $h_{j,-1}$ and
- differences in $k_{j,-1}$

The matrices Λ, Π, Δ_h, Θ_h do not depend on j.

This makes everybody's demand system have the form described earlier, with different μ_{j0}^w's (reflecting different wealth levels) and different b_{jt} preference shock processes and initial conditions for household capital stocks.

Punchline: there exists a representative consumer.

We can use the representative consumer to compute a competitive equilibrium **aggregate** allocation and price system.

With the equilibrium aggregate allocation and price system in hand, we can then compute allocations to each household.

Computing Allocations to Individuals:

Set

$$\ell_{jt} = (\mu_{0j}^w / \mu_{0a}^w) \ell_{at}$$

Then solve the following equation for μ_{0j}^w:

$$\mu_{0j}^w E_0 \sum_{t=0}^{\infty} \beta^t \{\rho_t^0 \cdot \rho_t^0 + (w_t^0 / \mu_{0a}^w) \ell_{at}\} = E_0 \sum_{t=0}^{\infty} \beta^t \{\rho_t^0 \cdot (b_{jt} - s_{jt}^i) - \alpha_t^0 \cdot d_{jt}\} - v_0 k_{j,-1}$$

$$s_{jt} - b_{jt} = \mu_{0j}^w \rho_t^0$$

$$\begin{aligned} c_{jt} &= -\Pi^{-1} \Lambda h_{j,t-1} + \Pi^{-1} s_{jt} \\ h_{jt} &= (\Delta_h - \Theta_h \Pi^{-1} \Lambda) h_{j,t-1} + \Pi^{-1} \Theta_h s_{jt} \end{aligned}$$

Here $h_{j,-1}$ given.

78.13 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

$$\begin{aligned} p_t = {} & (\mu_{01}\Pi_1^{-1}\Pi_1^{-1\prime} + \mu_{02}\Pi_2^{-1}\Pi_2^{-1\prime})^{-1}(\Pi_1^{-1} b_{1t} + \Pi_2^{-1} b_{2t}) \\ & - (\mu_{01}\Pi_1^{-1}\Pi_1^{-1\prime} + \mu_{02}\Pi_2^{-1}\Pi_2^{-1\prime})^{-1} c_t \end{aligned}$$

Punchline: choose Π associated with the aggregate ordering to satisfy

$$\mu_0^{-1}\Pi'\Pi = (\mu_{01}\Pi_1^{-1}\Pi_2^{-1\prime} + \mu_{02}\Pi_2^{-1}\Pi_2^{-1\prime})^{-1}$$

Dynamic Analogue:

We now describe how to extend mongrel aggregation to a dynamic setting.

The key comparison is

- Static: factor a covariance matrix-like object
- Dynamic: factor a spectral-density matrix-like object

Programming Problem for Dynamic Mongrel Aggregation:

Our strategy for deducing the mongrel preference ordering over $c_t = c_{1t} + c_{2t}$ is to solve the programming problem: choose $\{c_{1t}, c_{2t}\}$ to maximize the criterion

$$\sum_{t=0}^{\infty} \beta^t [\lambda(s_{1t} - b_{1t})\cdot(s_{1t} - b_{1t}) + (1-\lambda)(s_{2t} - b_{2t})\cdot(s_{2t} - b_{2t})]$$

subject to

$$\begin{aligned} h_{jt} &= \Delta_{hj} h_{jt-1} + \Theta_{hj} c_{jt}, j = 1, 2 \\ s_{jt} &= \Delta_j h_{jt-1} + \Pi_j c_{jt}\ , j = 1, 2 \\ c_{1t} + c_{2t} &= c_t \end{aligned}$$

subject to $(h_{1,-1},\, h_{2,-1})$ given and $\{b_{1t}\}$, $\{b_{2t}\}$, $\{c_t\}$ being known and fixed sequences.

Substituting the $\{c_{1t},\, c_{2t}\}$ sequences that solve this problem as functions of $\{b_{1t},\, b_{2t},\, c_t\}$ into the objective determines a mongrel preference ordering over $\{c_t\} = \{c_{1t} + c_{2t}\}$.

In solving this problem, it is convenient to proceed by using Fourier transforms. For details, please see [78] where they deploy a

Secret Weapon: Another application of the spectral factorization identity.

Concluding remark: The [78] 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 79

Growth in Dynamic Linear Economies

79.1 Contents

- Common Structure 79.2
- A Planning Problem 79.3
- Example Economies 79.4

Co-author: Sebastian Graves

This is another member of a suite of lectures that use the quantecon DLE class to instantiate models within the [78] 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.

```
In [1]: !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

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        from quantecon import LQ, DLE
```

79.2 Common Structure

Our example economies have the following features

- Information flows are governed by an exogenous stochastic process z_t that follows

$$z_{t+1} = A_{22} z_t + C_2 w_{t+1}$$

where w_{t+1} is a martingale difference sequence.

- Preference shocks b_t and technology shocks d_t are linear functions of z_t

$$b_t = U_b z_t$$
$$d_t = U_d z_t$$

- Consumption and physical investment goods are produced using the following technology

$$\Phi_c c_t + \Phi_g g_t + \Phi_i i_t = \Gamma k_{t-1} + d_t$$
$$k_t = \Delta_k k_{t-1} + \Theta_k i_t$$
$$g_t \cdot g_t = l_t^2$$

where c_t is a vector of consumption goods, g_t is a vector of intermediate goods, i_t is a vector of investment goods, k_t is a vector of physical capital goods, and l_t is the amount of labor supplied by the representative household.

- Preferences of a representative household are described by

$$-\frac{1}{2}\mathbb{E}\sum_{t=0}^{\infty}\beta^t[(s_t - b_t)\cdot(s_t - b_t) + l_t^2], 0 < \beta < 1$$
$$s_t = \Lambda h_{t-1} + \Pi c_t$$
$$h_t = \Delta_h h_{t-1} + \Theta_h c_t$$

where s_t is a vector of consumption services, and h_t is a vector of household capital stocks.

Thus, an instance of this class of economies is described by the matrices

$$\{A_{22}, C_2, U_b, U_d, \Phi_c, \Phi_g, \Phi_i, \Gamma, \Delta_k, \Theta_k, \Lambda, \Pi, \Delta_h, \Theta_h\}$$

and the scalar β.

79.3 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) [78] 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

$$x_{t+1} = A^o x_t + Cw_{t+1}$$

79.4 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.

79.4.1 Example 1: Hall (1978)

First, we set parameters such that consumption follows a random walk. In particular, we set

$$\lambda = 0, \pi = 1, \gamma_1 = 0.1, \phi_1 = 0.00001, \delta_k = 0.95, \beta = \frac{1}{1.05}$$

(In this economy δ_h and θ_h are arbitrary as household capital does not enter the equation for consumption services We set them to values that will become useful in Example 3)

It is worth noting that this choice of parameter values ensures that $\beta(\gamma_1 + \delta_k) = 1$.

For simulations of this economy, we choose an initial condition of

$$x_0 = \begin{bmatrix} 5 & 150 & 1 & 0 & 0 \end{bmatrix}'$$

In [3]:
```
# 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.

In [4]:
```
econ1 = DLE(info1, tech1, pref1)
```

We can then simulate the economy for a chosen length of time, from our initial state vector x_0

In [5]:
```
econ1.compute_sequence(x0, ts_length=300)
```

The economy stores the simulated values for each variable. Below we plot consumption and investment

In [6]:
```
# 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

```
In [7]: econ1.endo, econ1.exo
```

```
Out[7]: (array([0.9, 1. ]), array([1. , 0.8, 0.5]))
```

The endogenous eigenvalue that appears to be unity reflects the random walk character of consumption in Hall's model.

- Actually, the largest endogenous eigenvalue is very slightly below 1.
- This outcome comes from the small adjustment cost ϕ_1.

```
In [8]: econ1.endo[1]
```

```
Out[8]: 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.

```
In [9]: 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.

79.4.2 Example 2: Altered Growth Condition

We generate our next economy by making two alterations to the parameters of Example 1.

- First, we raise ϕ_1 from 0.00001 to 1.
  - This will lower the endogenous eigenvalue that is close to 1, causing the economy to head more quickly to the vicinity of its non-stochastic steady-state.
- Second, we raise γ_1 from 0.1 to 0.15.
  - This has the effect of raising the optimal steady-state value of capital.

We also start the economy off from an initial condition with a lower capital stock

$$x_0 = \begin{bmatrix} 5 & 20 & 1 & 0 & 0 \end{bmatrix}'$$

Therefore, we need to define the following new parameters

```
In [10]: 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

```
In [11]: 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

```
In [12]: 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.

```
In [13]: econ2.endo, econ2.exo
```

```
Out[13]: (array([0.9  , 0.952]), array([1. , 0.8, 0.5]))
```

79.4.3 Example 3: A Jones-Manuelli (1990) Economy

For our third economy, we choose parameter values with the aim of generating *sustained* growth in consumption, investment and capital.

To do this, we set parameters so that Jones and Manuelli's "growth condition" is just satisfied.

In our notation, just satisfying the growth condition is actually equivalent to setting $\beta(\gamma_1 + \delta_k) = 1$, the condition that was necessary for consumption to be a random walk in Hall's model.

Thus, we lower γ_1 back to 0.1.

In our model, this is a necessary but not sufficient condition for growth.

To generate growth we set preference parameters to reflect habit persistence.

In particular, we set $\lambda = -1$, $\delta_h = 0.9$ and $\theta_h = 1 - \delta_h = 0.1$.

This makes preferences assume the form

$$-\frac{1}{2}\mathbb{E}\sum_{t=0}^{\infty}\beta^t[(c_t - b_t - (1-\delta_h)\sum_{j=0}^{\infty}\delta_h^j c_{t-j-1})^2 + l_t^2]$$

These preferences reflect habit persistence

- the effective "bliss point" $b_t + (1-\delta_h)\sum_{j=0}^{\infty}\delta_h^j c_{t-j-1}$ now shifts in response to a moving average of past consumption

Since δ_h and θ_h were defined earlier, the only change we need to make from the parameters of Example 1 is to define the new value of λ.

```
In [14]: l_λ2 = np.array([[-1]])
         pref2 = (β, l_λ2, π_h, δ_h, θ_h)
```

```
In [15]: econ3 = DLE(info1, tech1, pref2)
```

We simulate this economy from the original state vector

```
In [16]: 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

```
In [17]: econ3.endo, econ3.exo
```

```
Out[17]: (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.

79.4.4 Example 3.1: Varying Sensitivity

Next we raise λ to -0.7

```
In [18]: l_λ3 = np.array([[-0.7]])
         pref3 = (β, l_λ3, π_h, δ_h, θ_h)

         econ4 = DLE(info1, tech1, pref3)

         econ4.compute_sequence(x0, ts_length=300)

         plt.plot(econ4.c[0], label='Cons.')
         plt.plot(econ4.i[0], label='Inv.')
         plt.legend()
         plt.show()
```

We no longer achieve sustained growth if λ is raised from -1 to -0.7.

This is related to the fact that one of the endogenous eigenvalues is now less than 1.

```
In [19]: econ4.endo, econ4.exo
```

```
Out[19]: (array([0.97, 1.  ]), array([1. , 0.8, 0.5]))
```

79.4.5 Example 3.2: More Impatience

Next let's lower β to 0.94

```
In [20]: β_2 = np.array([[0.94]])
         pref4 = (β_2, l_λ, π_h, δ_h, θ_h)

         econ5 = DLE(info1, tech1, pref4)

         econ5.compute_sequence(x0, ts_length=300)

         plt.plot(econ5.c[0], label='Cons.')
         plt.plot(econ5.i[0], label='Inv.')
         plt.legend()
         plt.show()
```

Growth also fails if we lower β, since we now have $\beta(\gamma_1 + \delta_k) < 1$.

Consumption and investment explode downwards, as a lower value of β causes the representative consumer to front-load consumption.

This explosive path shows up in the second endogenous eigenvalue now being larger than one.

```
In [21]: econ5.endo, econ5.exo
```

```
Out[21]: (array([0.9  , 1.013]), array([1. , 0.8, 0.5]))
```

Chapter 80

Lucas Asset Pricing Using DLE

80.1 Contents

- Asset Pricing Equations 80.2
- Asset Pricing Simulations 80.3

Co-author: Sebastian Graves

This is one of a suite of lectures that use the quantecon DLE class to instantiate models within the [78] 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

```
In [1]: !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

```
In [2]: 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) [109] used to develop an equilibrium theory of asset prices:

Preferences

$$-\frac{1}{2}\mathbb{E}\sum_{t=0}^{\infty}\beta^t[(c_t-b_t)^2+l_t^2]|J_0$$

$$s_t = c_t$$

$$b_t = U_b z_t$$

Technology

$$c_t = d_{1t}$$

$$k_t = \delta_k k_{t-1} + i_t$$

$$g_t = \phi_1 i_t \, , \phi_1 > 0$$

$$\begin{bmatrix} d_{1t} \\ 0 \end{bmatrix} = U_d z_t$$

Information

$$z_{t+1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.8 & 0 \\ 0 & 0 & 0.5 \end{bmatrix} z_t + \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} w_{t+1}$$

$$U_b = \begin{bmatrix} 30 & 0 & 0 \end{bmatrix}$$

$$U_d = \begin{bmatrix} 5 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$x_0 = \begin{bmatrix} 5 & 150 & 1 & 0 & 0 \end{bmatrix}'$$

80.2 Asset Pricing Equations

[78] 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.

80.3 Asset Pricing Simulations

```
In [3]: gam = 0
        γ = np.array([[gam], [0]])
        ϕ_c = np.array([[1], [0]])
        ϕ_g = np.array([[0], [1]])
        ϕ_1 = 1e-4
        ϕ_i = np.array([[0], [-ϕ_1]])
        δ_k = np.array([[.95]])
        θ_k = np.array([[1]])
        β = np.array([[1 / 1.05]])
        ud = np.array([[5, 1, 0],
                       [0, 0, 0]])
        a22 = np.array([[1,   0,    0],
                        [0, 0.8,    0],
                        [0,   0, 0.5]])
        c2 = np.array([[0, 1, 0],
                       [0, 0, 1]]).T
        l_λ = np.array([[0]])
        π_h = np.array([[1]])
        δ_h = np.array([[.9]])
        θ_h = np.array([[1]]) - δ_h
        ub = np.array([[30, 0, 0]])
        x0 = np.array([[5, 150, 1, 0, 0]]).T

        info1 = (a22, c2, ub, ud)
        tech1 = (ϕ_c, ϕ_g, ϕ_i, γ, δ_k, θ_k)
        pref1 = (β, l_λ, π_h, δ_h, θ_h)
```

```
In [4]: 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}

```
In [5]: econ1.compute_sequence(x0, ts_length=100, Pay=np.array([econ1.Sd[0, :]]))
```

The graph below plots the price of this claim over time:

```
In [6]: ### 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:

```
In [7]: ### 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()
```

```
In [8]: np.corrcoef(econ1.Pay_Gross[1:, 0], econ1.R1_Gross[1:, 0])
```

```
Out[8]: array([[ 1.        , -0.49650074],
               [-0.49650074,  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:

```
In [9]: ### 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:

```
In [10]: 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, :]]))
```

```
In [11]: ### 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()
```

```
In [12]: np.corrcoef(econ2.Pay_Gross[1:, 0], econ2.R1_Gross[1:, 0])
```

```
Out[12]: array([[ 1.        , -0.61977714],
                [-0.61977714,  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:

```
In [13]: ### 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 81

IRFs in Hall Models

81.1 Contents

- Example 1: Hall (1978) 81.2
- Example 2: Higher Adjustment Costs 81.3
- Example 3: Durable Consumption Goods 81.4

Co-author: Sebastian Graves

This is another member of a suite of lectures that use the quantecon DLE class to instantiate models within the [78] 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.

```
In [1]: !pip install --upgrade quantecon
```

We'll make these imports:

```
In [2]: 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) [67].

Knowledge of the basic economic environment is assumed.

See the lecture "Growth in Dynamic Linear Economies" for more details.

81.2 Example 1: Hall (1978)

First, we set parameters to make consumption (almost) follow a random walk.

We set

$$\lambda = 0, \pi = 1, \gamma_1 = 0.1, \phi_1 = 0.00001, \delta_k = 0.95, \beta = \frac{1}{1.05}$$

(In this example δ_h and θ_h are arbitrary as household capital does not enter the equation for consumption services.

We set them to values that will become useful in Example 3)

It is worth noting that this choice of parameter values ensures that $\beta(\gamma_1 + \delta_k) = 1$.

For simulations of this economy, we choose an initial condition of:

$$x_0 = \begin{bmatrix} 5 & 150 & 1 & 0 & 0 \end{bmatrix}'$$

In [3]:
```
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:

In [4]:
```
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:

```
In [5]: 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.

81.3 Example 2: Higher Adjustment Costs

We generate our next economy by making only one change to the parameters of Example 1: we raise the parameter associated with the cost of adjusting capital,ϕ_1, from 0.00001 to 0.2.

This will lower the endogenous eigenvalue that is unity in Example 1 to a value slightly below 1.

```
In [6]: 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()
```

```
In [7]: 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()
```

```
In [8]: econ2.endo
```

```
Out[8]: array([0.9       , 0.99657126])
```

```
In [9]: 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.

his 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.

81.4 Example 3: Durable Consumption Goods

We generate our third economy by raising ϕ_1 further, to 1.0. We also raise the production function parameter from 0.1 to 0.15 (which raises the non-stochastic steady state value of capital above zero).

We also change the specification of preferences to make the consumption good *durable*.

Specifically, we allow for a single durable household good obeying:

$$h_t = \delta_h h_{t-1} + c_t \,, 0 < \delta_h < 1$$

Services are related to the stock of durables at the beginning of the period:

$$s_t = \lambda h_{t-1} \,, \lambda > 0$$

And preferences are ordered by:

$$-\frac{1}{2}\mathbb{E}\sum_{t=0}^{\infty}\beta^t[(\lambda h_{t-1} - b_t)^2 + l_t^2]|J_0$$

To implement this, we set $\lambda = 0.1$ and $\pi = 0$ (we have already set $\theta_h = 1$ and $\delta_h = 0.9$).

We start from an initial condition that makes consumption begin near around its non-stochastic steady state.

```
In [10]: ϕ_13 = 1
         ϕ_i3 = np.array([[1], [-ϕ_13]])
```

```
γ_12 = 0.15
γ_2 = np.array([[γ_12], [0]])

l_λ2 = np.array([[0.1]])
π_h2 = np.array([[0]])

x01 = np.array([[150], [100], [1], [0], [0]])

tech3 = (ϕ_c, ϕ_g, ϕ_i3, γ_2, δ_k, θ_k)
pref2 = (β, l_λ2, π_h2, δ_h, θ_h)

econ3 = DLE(info1, tech3, pref2)
econ3.compute_sequence(x01, ts_length=300)

# This is the right panel of Fig 5.11.1 from p.111 of HS2013
plt.plot(econ3.c[0], label='Cons.')
plt.plot(econ3.i[0], label='Inv.')
plt.legend()
plt.show()
```

In contrast to Hall's original model of Example 1, it is now investment that is much smoother than consumption.

This illustrates how making consumption goods durable tends to undo the strong consumption smoothing result that Hall obtained.

```
In [11]: 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 82

Permanent Income Model using the DLE Class

82.1 Contents

- The Permanent Income Model 82.2

Co-author: Sebastian Graves

This lecture is part of a suite of lectures that use the quantecon DLE class to instantiate models within the [78] 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.

```
In [1]: !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 `http://nbviewer.jupyter.org/github/QuantEcon/QuantEcon.notebooks/blob/master/permanent_income.ipynb`.

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" [78].

We'll also require the following imports

```
In [2]: 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)
```

82.2 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

$$E_0 \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{1}$$

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 (1) by choosing a consumption, borrowing plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = R^{-1} b_{t+1} + y_t, t \geq 0 \tag{2}$$

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 (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

$$\begin{aligned} z_{t+1} &= A_{22} z_t + C_2 w_{t+1} \\ y_t &= U_y z_t \end{aligned} \tag{3}$$

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:

$$E_0 \sum_{t=0}^{\infty} \beta^t b_t^2 < +\infty \tag{4}$$

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:

$$y_{t+1} = \alpha + \rho_1 y_t + \rho_2 y_{t-1} + \sigma w_{t+1}$$

82.2.1 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 [78] by setting up our technology, information and preference matrices as follows:

Technology: $\phi_c = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $\phi_g = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $\phi_i = \begin{bmatrix} -1 \\ -0.00001 \end{bmatrix}$, $\Gamma = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$, $\Delta_k = 0$, $\Theta_k = R$.

Information: $A_{22} = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}$, $C_2 = \begin{bmatrix} 0 \\ \sigma \\ 0 \end{bmatrix}$, $U_b = \begin{bmatrix} \gamma & 0 & 0 \end{bmatrix}$, $U_d = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$.

Preferences: $\Lambda = 0$, $\Pi = 1$, $\Delta_h = 0$, $\Theta_h = 0$.

We set parameters

$\alpha = 10, \beta = 0.95, \rho_1 = 0.9, \rho_2 = 0, \sigma = 1$

(The value of γ does not affect the optimal decision rule)

The chosen matrices mean that the household's technology is:

$$c_t + k_{t-1} = i_t + y_t$$

$$\frac{k_t}{R} = i_t$$

$$l_t^2 = (0.00001)^2 i_t$$

Combining the first two of these gives the budget constraint of the permanent income model, where $k_t = b_{t+1}$.

The third equation is a very small penalty on debt-accumulation to rule out Ponzi schemes.

We set up this instance of the DLE class below:

```
In [3]: α, β, ρ_1, ρ_2, σ = 10, 0.95, 0.9, 0, 1

        γ = np.array([[-1], [0]])
        ϕ_c = np.array([[1], [0]])
        ϕ_g = np.array([[0], [1]])
        ϕ_1 = 1e-5
        ϕ_i = np.array([[-1], [-ϕ_1]])
        δ_k = np.array([[0]])
```

```
θ_k = np.array([[1 / β]])
β = np.array([[β]])
l_λ = np.array([[0]])
π_h = np.array([[1]])
δ_h = np.array([[0]])
θ_h = np.array([[0]])

a22 = np.array([[1,   0,   0],
                [α, ρ_1, ρ_2],
                [0, 1, 0]])

c2 = np.array([[0], [σ], [0]])
ud = np.array([[0, 1, 0],
               [0, 0, 0]])
ub = np.array([[100, 0, 0]])

x0 = np.array([[0], [0], [1], [0], [0]])

info1 = (a22, c2, ub, ud)
tech1 = (ϕ_c, ϕ_g, ϕ_i, γ, δ_k, θ_k)
pref1 = (β, l_λ, π_h, δ_h, θ_h)
econ1 = DLE(info1, tech1, pref1)
```

To check the solution of this model with that from the **LQ** problem, we select the S_c matrix from the DLE class.

The solution to the DLE economy has:

$$c_t = S_c x_t$$

```
In [4]: econ1.Sc
```

```
Out[4]: 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 http://nbviewer.jupyter.org/github/QuantEcon/QuantEcon.notebooks/blob/master/permanent_income.ipynb.

The plot below quickly replicates the first two figures of that lecture and that notebook to confirm that the solutions are the same

```
In [5]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 5))
```

```
for i in range(25):
    econ1.compute_sequence(x0, ts_length=150)
    ax1.plot(econ1.c[0], c='g')
    ax1.plot(econ1.d[0], c='b')
ax1.plot(econ1.c[0], label='Consumption', c='g')
ax1.plot(econ1.d[0], label='Income', c='b')
ax1.legend()

for i in range(25):
    econ1.compute_sequence(x0, ts_length=150)
    ax2.plot(econ1.k[0], color='r')
ax2.plot(econ1.k[0], label='Debt', c='r')
ax2.legend()
plt.show()
```

Chapter 83

Rosen Schooling Model

83.1 Contents

- A One-Occupation Model 83.2
- Mapping into HS2013 Framework 83.3

Co-author: Sebastian Graves

This lecture is yet another part of a suite of lectures that use the quantecon DLE class to instantiate models within the [78] 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

```
In [1]: !pip install --upgrade quantecon
```

We'll also need the following imports:

```
In [2]: 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
```

83.2 A One-Occupation Model

Ryoo and Rosen's (2004) [138] 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:

$$n_t = \alpha_s v_t + \epsilon_{st}$$

83.3 Mapping into HS2013 Framework

We represent this model in the [78] 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

83.3.1 Preferences

$$\Pi = 0, \Lambda = \begin{bmatrix} \alpha_d & 0 & \cdots & 0 \end{bmatrix}, \Delta_h = \begin{bmatrix} \delta_N & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & 1 \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix}, \Theta_h = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

where Λ is a k+1 x 1 matrix, Δ_h is a k_1 x k+1 matrix, and Θ_h is a k+1 x 1 matrix.

This specification sets $N_t = h_{1t-1}$, $n_t = c_t$, $h_{\tau+1,t-1} = n_{t-(k-\tau)}$ for $\tau = 1, ..., k$.

Below we set things up so that the number of years of education, k, can be varied.

83.3.2 Technology

To capture Ryoo and Rosen's [138] supply curve, we use the physical technology:

$$c_t = i_t + d_{1t}$$

$$\psi_1 i_t = g_t$$

where ψ_1 is inversely proportional to α_s.

83.3.3 Information

Because we want $b_t = \epsilon_{dt}$ and $d_{1t} = \epsilon_{st}$, we set

$$A_{22} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \rho_s & 0 \\ 0 & 0 & \rho_d \end{bmatrix}, C_2 = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, U_b = \begin{bmatrix} 30 & 0 & 1 \end{bmatrix}, U_d = \begin{bmatrix} 10 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

where ρ_s and ρ_d describe the persistence of the supply and demand shocks

```
In [3]: 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'])
```

83.3.4 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 $\alpha_d = 0.1$

```
In [4]: 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.ones((1, k)) * ε_1
        # Use of ε_1 is trick to aquire detectability, see HS2013 p. 228 footnote 4
        l_λ = np.hstack((α_d, λ_1))
        π_h = np.array([[0]])

        δ_n = np.array([[0.95]])
        d1 = np.vstack((δ_n, np.zeros((k - 1, 1))))
        d2 = np.hstack((d1, np.eye(k)))
        δ_h = np.vstack((d2, np.zeros((1, k + 1))))

        θ_h = np.vstack((np.zeros((k, 1)),
                         np.ones((1, 1))))

        ψ_1 = 1 / α_s

        ϕ_c = np.array([[1], [0]])
        ϕ_g = np.array([[0], [-1]])
        ϕ_i = np.array([[-1], [ψ_1]])
        γ = np.array([[0], [0]])

        δ_k = np.array([[0]])
        θ_k = np.array([[0]])

        ρ_s = 0.8
```

```
ρ_d = 0.8

a22 = np.array([[1,   0,   0],
                [0, ρ_s,   0],
                [0,   0, ρ_d]])

c2 = np.array([[0, 0], [10, 0], [0, 10]])
ub = np.array([[30, 0, 1]])
ud = np.array([[10, 1, 0], [0, 0, 0]])

info1 = Information(a22, c2, ub, ud)
tech1 = Technology(ϕ_c, ϕ_g, ϕ_i, γ, δ_k, θ_k)
pref1 = Preferences(β, l_λ, π_h, δ_h, θ_h)

econ1 = DLE(info1, tech1, pref1)
```

We create three other instances by:

1. Raising α_d to 2
2. Raising k to 7
3. Raising k to 10

In [5]:
```
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.ones((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.ones((1, k)) * ε_1
l_λ = np.hstack((α_d, λ_1))
d1 = np.vstack((δ_n, np.zeros((k - 1, 1))))
d2 = np.hstack((d1, np.eye(k)))
δ_h = np.vstack((d2, np.zeros((1, k + 1))))
θ_h = np.vstack((np.zeros((k, 1)),
                 np.ones((1, 1))))

pref4 = Preferences(β, l_λ, π_h, δ_h, θ_h)
econ4 = DLE(info1, tech1, pref4)

shock_demand = np.array([[0], [1]])
```

```
econ1.irf(ts_length=25, shock=shock_demand)
econ2.irf(ts_length=25, shock=shock_demand)
econ3.irf(ts_length=25, shock=shock_demand)
econ4.irf(ts_length=25, shock=shock_demand)
```

The first figure plots the impulse response of n_t (on the left) and N_t (on the right) to a positive demand shock, for $\alpha_d = 0.1$ and $\alpha_d = 2$.

When $\alpha_d = 2$, the number of new students n_t rises initially, but the response then turns negative.

A positive demand shock raises wages, drawing new students into the profession.

However, these new students raise N_t.

The higher is α_d, the larger the effect of this rise in N_t on wages.

This counteracts the demand shock's positive effect on wages, reducing the number of new students in subsequent periods.

Consequently, when α_d is lower, the effect of a demand shock on N_t is larger

In [6]:
```
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 $\alpha_d = 0.1$)

In [7]:
```
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 84

Cattle Cycles

84.1 Contents

- The Model 84.2
- Mapping into HS2013 Framework 84.3

Co-author: Sebastian Graves

This is another member of a suite of lectures that use the quantecon DLE class to instantiate models within the [78] 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.

```
In [1]: !pip install --upgrade quantecon
```

This lecture uses the DLE class to construct instances of the "Cattle Cycles" model of Rosen, Murphy and Scheinkman (1994) [133].

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:

```
In [2]: 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
```

84.2 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 ψ_j parameters are very small quadratic costs that are included for technical reasons to make well posed and well behaved the linear quadratic dynamic programming problem solved by the fictitious planner who in effect chooses equilibrium quantities and shadow prices.

Demand for beef is government by $c_t = a_0 - a_1 p_t + \tilde{d}_t$ where $\tilde{d}_t$ is a stochastic process with mean zero, representing a demand shifter.

84.3 Mapping into HS2013 Framework

84.3.1 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.

84.3.2 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

$$\Phi_c = \begin{bmatrix} 1 \\ f_1 \\ 0 \\ 0 \\ -f_7 \end{bmatrix}, \Phi_g = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \Phi_i = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \Gamma = \begin{bmatrix} 0 & 0 & 0 \\ f_1(1-\delta) & 0 & gf_1 \\ f_3 & 0 & 0 \\ 0 & f_5 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

84.3.3 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 $f_1^2 = \frac{\Psi_1}{2}$, $f_2^2 = \frac{\Psi_2}{2}$, $f_3^2 = \frac{\Psi_3}{2}$, $2f_1f_2 = 1$, $2f_3f_4 = \gamma_0 g$, $2f_5f_6 = \gamma_1 g$.

```
In [3]: # 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 [133]

```
In [4]: β = np.array([[0.909]])
        lλ = np.array([[0]])

        a1 = 0.5
        πh = np.array([[1 / (sqrt(a1))]])
        δh = np.array([[0]])
        θh = np.array([[0]])

        δ = 0.1
        g = 0.85
        f1 = 0.001
        f3 = 0.001
        f5 = 0.001
        f7 = 0.001

        ϕc = np.array([[1], [f1], [0], [0], [-f7]])

        ϕg = np.array([[0, 0, 0, 0],
                       [1, 0, 0, 0],
```

```
                [0, 1, 0, 0],
                [0, 0, 1,0],
                [0, 0, 0, 1]])

ϕi = np.array([[1], [0], [0], [0], [0]])

γ = np.array([[           0,  0,      0],
              [f1 * (1 - δ),  0, g * f1],
              [          f3,  0,      0],
              [           0, f5,      0],
              [           0,  0,      0]])

δk = np.array([[1 - δ, 0, g],
               [    1, 0, 0],
               [    0, 1, 0]])

θk = np.array([[1], [0], [0]])

ρ1 = 0
ρ2 = 0
ρ3 = 0.6
a0 = 500
γ0 = 0.4
γ1 = 0.7
f2 = 1 / (2 * f1)
f4 = γ0 * g / (2 * f3)
f6 = γ1 * g / (2 * f5)
f8 = 1 / (2 * f7)

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))
```

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 $\rho_3 = 0.6$).

```
In [5]: info1 = Information(a22, c2, ub, ud)
        tech1 = Technology(ϕc, ϕg, ϕi, γ, δk, θk)
        pref1 = Preferences(β, lλ, πh, δh, θh)
```

```
ρ3_2 = 1
a22_2 = np.array([[1,  0,  0,    0],
                  [0, ρ1,  0,    0],
                  [0,  0, ρ2,    0],
                  [0,  0,  0, ρ3_2]])

info2 = Information(a22_2, c2, ub, ud)

ρ3_3 = 0
a22_3 = np.array([[1,  0,  0,    0],
                  [0, ρ1,  0,    0],
                  [0,  0, ρ2,    0],
                  [0,  0,  0, ρ3_3]])

info3 = Information(a22_3, c2, ub, ud)

# Example of how we can look at the matrices associated with a given
↪namedtuple
info1.a22
```

```
Out[5]: array([[1. , 0. , 0. , 0. ],
               [0. , 0. , 0. , 0. ],
               [0. , 0. , 0. , 0. ],
               [0. , 0. , 0. , 0.6]])
```

```
In [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
```

```
In [7]: econ1.compute_sequence(x0, ts_length=100)
```

[133] 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

```
In [8]: # 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, [133] plot the impulse response functions of consumption and the breeding stock of cattle to the demand shock, $\tilde{d}_t$, under the three different values of ρ_3.

We replicate their Figure 3 below

```
In [9]: 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='$\\rho=0.6$')
        ax1.plot(econ2.c_irf, label='$\\rho=1$')
        ax1.plot(econ3.c_irf, label='$\\rho=0$')
        ax1.set_title('Consumption response to demand shock')
        ax1.legend()

        ax2.plot(econ1.k_irf[:, 0], label='$\\rho=0.6$')
        ax2.plot(econ2.k_irf[:, 0], label='$\\rho=1$')
        ax2.plot(econ3.k_irf[:, 0], label='$\\rho=0$')
        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, [133] 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

```
In [10]: 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('$\\rho=0.6$')

         ax2.plot(econ3.k_irf[:, 0], label='Breeding Stock')
         ax2.plot(total3_irf, label='Total Stock')
         ax2.set_title('$\\rho=0$')
         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 85

Shock Non Invertibility

85.1 Contents

- Overview 85.2
- Model 85.3
- Code 85.4

85.2 Overview

This is another member of a suite of lectures that use the quantecon DLE class to instantiate models within the [78] 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.

```
In [1]: !pip install --upgrade quantecon
```

We'll make these imports:

```
In [2]: 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 [**?**] in their analysis of **fiscal foresight**.

A distinct shock-invertibility issue is present in the special LQ consumption smoothing model in quantecon lecture.

85.3 Model

We consider the following modification of Robert Hall's (1978) model [67] 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.

In [3]:
```
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_ts to be:

$$\begin{bmatrix} c_t \\ c_t - d_t \end{bmatrix} = \begin{bmatrix} \sigma_1(L) \\ \sigma_2(L) \end{bmatrix} w_t$$

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 [78] 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_ts spans a *smaller* linear space than does the history of w_ts.

This means that u_t will typically be a distributed lag of w_t that is not concentrated at zero lag:

$$u_t = \sum_{j=0}^{\infty} \alpha_j w_{t-j}$$

Thus, the econometrician's news u_t potentially responds belatedly to agents' news w_t.

85.4 Code

We will construct Figures from Chapter 8 Appendix E of [78] to illustrate these ideas:

In [4]:
```
# 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.

In [5]:
```
G_HS = np.vstack([econ1.Sc, econ1.Sc-econ1.Sd[0, :].reshape(1, 8)])
H_HS = 1e-8 * np.eye(2)  # Set very small so there is no measurement error
lss_hs = qe.LinearStateSpace(econ1.A0, econ1.C, G_HS, H_HS)

hs_kal = qe.Kalman(lss_hs)
w_lss = hs_kal.whitener_lss()
ma_coefs = hs_kal.stationary_coefficients(50, 'ma')

# This is Fig 8.E.2 from p.189 of HS2013

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.

In [6]:
```
# 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 XI

Classic Linear Models

Chapter 86

Von Neumann Growth Model (and a Generalization)

86.1 Contents

- Notation 86.2
- Model Ingredients and Assumptions 86.3
- Dynamic Interpretation 86.4
- Duality 86.5
- Interpretation as a Game Theoretic Problem (Two-player Zero-sum Game) 86.6

Co-author: Balint Szoke

This notebook uses the class `Neumann` to calculate key objects of a linear growth model of John von Neumann [161] that was generalized by Kemeny, Morgenstern and Thompson [96].

Objects of interest are the maximal expansion rate (α), the interest factor (β), and 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 notebook 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:

In [1]:
```
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

In [2]:
```
class Neumann(object):
```

```
    """
    This class describes the Generalized von Neumann growth model as it was
    discussed in Kemeny et al. (1956, ECTA) and Gale (1960, Chapter 9.5):

    Let:
    n ... number of goods
    m ... number of activities
    A ... input matrix is m-by-n
        a_{i,j} - amount of good j consumed by activity i
    B ... output matrix is m-by-n
        b_{i,j} - amount of good j produced by activity i

    x ... intensity vector (m-vector) with non-negative entries
        x'B - the vector of goods produced
        x'A - the vector of goods consumed
    p ... price vector (n-vector) with non-negative entries
        Bp - the revenue vector for every activity
        Ap - the cost of each activity

    Both A and B have non-negative entries. Moreover, we assume that
    (1) Assumption I (every good which is consumed is also produced):
       for all j, b_{.,j} > 0, i.e. at least one entry is strictly positive
    (2) Assumption II (no free lunch):
       for all i, a_{i,.} > 0, i.e. at least one entry is strictly positive

    Parameters
    ----------
    A : array_like or scalar(float)
        Part of the state transition equation.  It should be `n x n`
    B : array_like or scalar(float)
        Part of the state transition equation.  It should be `n x k`
    """

    def __init__(self, A, B):

        self.A, self.B = list(map(self.convert, (A, B)))
        self.m, self.n = self.A.shape

        # Check if (A, B) satisfy the basic assumptions
       assert self.A.shape == self.B.shape, 'The input and output matrices \
               must have the same dimensions!'
        assert (self.A >= 0).all() and (self.B >= 0).all(), 'The input and \
               output matrices must have only non-negative entries!'

        # (1) Check whether Assumption I is satisfied:
        if (np.sum(B, 0) <= 0).any():
            self.AI = False
        else:
            self.AI = True

        # (2) Check whether Assumption II is satisfied:
        if (np.sum(A, 1) <= 0).any():
            self.AII = False
        else:
            self.AII = True

    def __repr__(self):
```

```
        return self.__str__()

    def __str__(self):

        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 = np.asscalar(fsolve(f, 1))  # Upper bound for α, β
        LB = np.asscalar(fsolve(g, 2))  # Lower bound for α, β

        return LB, UB


    def zerosum(self, γ, dual=False):
        """
        Given gamma, calculate the value and optimal strategies of a
        two-player zero-sum game given by the matrix

                M(gamma) = B - gamma * A

        Row player maximizing, column player minimizing

        Zero-sum game as an LP (primal --> α)

            max (0', 1) @ (x', v)
```

```
            subject to
            [-M', ones(n, 1)] @ (x', v)' <= 0
            (x', v) @ (ones(m, 1), 0) = 1
            (x', v) >= (0', -inf)

        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, options=dict(bland=True, tol=1e-7))

        else:
            # Solve the dual LP (for details see the description)
            # (1) Define the problem for v as a maximization (linprog
↪minimizes)
            c = np.hstack([np.zeros(n), 1])

            # (2) Add constraints :
            # ... non-negativity constraints
            bounds = tuple(n * [(0, None)] + [(None, None)])
            # ... inequality constraints
            A_iq = np.hstack([M, -np.ones((m, 1))])
            b_iq = np.zeros((m, 1))
            # ... normalization
```

```
        A_eq = np.hstack([np.ones(n), 0]).reshape(1, n + 1)
        b_eq = 1

        res = linprog(c, A_ub=A_iq, b_ub=b_iq, A_eq=A_eq, b_eq=b_eq,
                      bounds=bounds, options=dict(bland=True, tol=1e-7))

    if res.status != 0:
        print(res.message)

    # Pull out the required quantities
    value = res.x[-1]
    strategy = res.x[:-1]

    return value, strategy


def expansion(self, tol=1e-8, maxit=1000):
    """
    The algorithm used here is described in Hamburger-Thompson-Weil
   (1967, ECTA). It is based on a simple bisection argument and utilizes
    the idea that for a given γ (= α or β), the matrix "M = B - γ * A"
   defines a two-player zero-sum game, where the optimal strategies are
    the (normalized) intensity and price vector.

    Outputs:
    --------
    alpha: scalar
        optimal expansion rate
    """

    LB, UB = self.bounds()

    for iter in range(maxit):

        γ = (LB + UB) / 2
        ZS = self.zerosum(γ=γ)
        V = ZS[0]     # value of the game with γ

        if V >= 0:
            LB = γ
        else:
            UB = γ

        if abs(UB - LB) < tol:
            γ = (UB + LB) / 2
            x = self.zerosum(γ=γ)[1]
            p = self.zerosum(γ=γ, dual=True)[1]
            break

    return γ, x, p

def interest(self, tol=1e-8, maxit=1000):
    """
    The algorithm used here is described in Hamburger-Thompson-Weil
   (1967, ECTA). It is based on a simple bisection argument and utilizes
    the idea that for a given gamma (= alpha or beta),
    the matrix "M = B - γ * A" defines a two-player zero-sum game,
   where the optimal strategies are the (normalized) intensity and price
```

```
        vector

        Outputs:
        --------
        beta: scalar
            optimal interest rate
        """

        LB, UB = self.bounds()

        for iter in range(maxit):
            γ = (LB + UB) / 2
            ZS = self.zerosum(γ=γ, dual=True)
            V = ZS[0]

            if V > 0:
                LB = γ
            else:
                UB = γ

            if abs(UB - LB) < tol:
                γ = (UB + LB) / 2
                p = self.zerosum(γ=γ, dual=True)[1]
                x = self.zerosum(γ=γ)[1]
                break

        return γ, x, p
```

86.2 Notation

We use the following notation.

$\mathbf{0}$ denotes a vector of zeros. We call an n-vector - positive or $x \gg \mathbf{0}$ if $x_i > 0$ for all $i = 1, 2, \dots, n$ - non-negative or $x \geq \mathbf{0}$ if $x_i \geq 0$ for all $i = 1, 2, \dots, n$ - semi-positive or $x > \mathbf{0}$ 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 $x - y > \mathbf{0}$.

By default, all vectors are column vectors, x^T denotes the transpose of x (i.e. a row vector).

Let ι_n denote a column vector composed of n ones, i.e. $\iota_n = (1, 1, \dots, 1)^T$.

Let e^i denote the 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.

86.3 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 used in the economy)
- 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 which is consumed is also produced)

$$b_{.,j} > \mathbf{0} \quad \forall j = 1, 2, \dots, n$$

- **Assumption II:** (no free lunch)

$$a_{i,.} > \mathbf{0} \quad \forall i = 1, 2, \dots, m$$

A semi-positive m-vector:math:x denotes the levels at which activities are operated (*intensity vector*).

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

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: Given an economy (A, B), the set of goods $S \subset \{1, 2, \dots, n\}$ is called an *independent subset* if it is possible to produce every good in S without consuming any good outside S. Formally, the set S is independent if $\exists T \subset \{1, 2, \dots, m\}$ (subset of activities) such that $a_{i,j} = 0$, $\forall i \in T$ and $j \in S^c$ and for all $j \in S$, $\exists i \in T$, s.t. $b_{i,j} > 0$. The economy is **irreducible** if there are no proper independent subsets.

We study two examples, both coming from Chapter 9.6 of Gale [60]

In [3]:
```
# (1) Irreducible (A, B) example: α_0 = β_0
A1 = np.array([[0, 1, 0, 0],
               [1, 0, 0, 1],
               [0, 0, 1, 0]])

B1 = np.array([[1, 0, 0, 0],
               [0, 0, 2, 0],
               [0, 1, 0, 1]])

# (2) Reducible (A, B) example: β_0 < α_0
A2 = np.array([[0, 1, 0, 0, 0, 0],
               [1, 0, 1, 0, 0, 0],
               [0, 0, 0, 1, 0, 0],
               [0, 0, 1, 0, 0, 1],
```

```
                [0, 0, 0, 0, 1, 0]])

B2 = np.array([[1, 0, 0, 1, 0, 0],
                [0, 1, 0, 0, 0, 0],
                [0, 0, 1, 0, 0, 0],
                [0, 0, 0, 0, 2, 0],
                [0, 0, 0, 1, 0, 1]])
```

The following code sets up our first Neumann economy or `Neumann` instance

In [4]:
```
n1 = Neumann(A1, B1)
n1
```

Out[4]:
```
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
```

In [5]:
```
n2 = Neumann(A2, B2)
n2
```

Out[5]:
```
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
```

86.4 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 notebook, 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 associated 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 $t+1$.

86.4.1 Balanced Growth

We follow John von Neumann in studying "balanced growth".

Let ./ denote an elementwise division of one vector by another and let $\alpha > 0$ be a scalar.

Then *balanced growth* is a situation in which

$$x_{t+1}./x_t = \alpha, \quad \forall t \geq 0$$

With balanced growth, the law of motion of x is evidently $x_{t+1} = \alpha x_t$ and so we can rewrite the feasibility constraint as

$$x_t^T B \geq \alpha x_t^T A \qquad \forall t$$

In the same spirit, define $\beta \in \mathbb{R}$ as the **interest factor** per unit of time.

We assume that it is always possible to earn a gross return equal to the constant interest factor β by investing "outside the model".

Under this assumption about outside investment opportunities, a no-arbitrage condition gives rise to the following (no profit) restriction on the price sequence:

$$\beta A p_t \geq B p_t \qquad \forall t$$

This says that production cannot yield a return greater than that offered by the investment opportunity (note that we compare values in period $t+1$).

The balanced growth assumption allows us to drop time subscripts and conduct an analysis purely in terms of a time-invariant growth rate α and interest factor β.

86.5 Duality

The following two problems are connected by a remarkable dual relationship between the 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 $\alpha \in \mathbb{R}$, s.t.

$$\begin{aligned} \max_{\alpha} \quad & \alpha \\ \text{s.t.} \quad & x^T B \geq \alpha x^T A \end{aligned}$$

Theorem 9.3 of David Gale's book [60] asserts that if Assumptions I and II are both satisfied, then a maximum value of α exists and it is positive.

It is called the *technological expansion rate* and is denoted by α_0. The associated intensity vector x_0 is the *optimal intensity vector*.

Definition: The *economical expansion problem* (EEP) for (A, B) is to find a semi-positive n-vector $p > 0$ and a number $\beta \in \mathbb{R}$, such that

$$\begin{aligned} \min_{\beta} \quad & \beta \\ \text{s.t.} \quad & Bp \leq \beta Ap \end{aligned}$$

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*.

Evidently, the criterion functions in *technological expansion* problem and the *economical expansion problem* are both linearly homogeneous, so the optimality of x_0 and p_0 are defined only up to a positive scale factor.

For simplicity (and to emphasize a close connection to zero-sum games), in the following, we normalize both vectors x_0 and p_0 to have unit length.

A standard duality argument (see Lemma 9.4. in (Gale, 1960) [60]) implies that under Assumptions I and II, $\beta_0 \leq \alpha_0$.

But in the other direction, that is $\beta_0 \geq \alpha_0$, Assumptions I and II are not sufficient.

Nevertheless, von Neumann [161] proved the following remarkable "duality-type" result connecting TEP and EEP.

Theorem 1 (von Neumann): If the economy (A, B) satisfies Assumptions I and II, then there exists a set (γ^*, x_0, p_0), where $\gamma^* \in [\beta_0, \alpha_0] \subset \mathbb{R}$, $x_0 > 0$ is an m-vector, $p_0 > 0$ is an n-vector and the following holds true

$$\begin{aligned} x_0^T B &\geq \gamma^* x_0^T A \\ Bp_0 &\leq \gamma^* Ap_0 \\ x_0^T \left(B - \gamma^* A\right) p_0 &= 0 \end{aligned}$$

Proof (Sketch): Assumption I and II imply that there exist (α_0, x_0) and (β_0, p_0) solving the TEP and EEP, respectively. If $\gamma^* > \alpha_0$, then by definition of α_0, there cannot exist a semi-positive x that satisfies $x^T B \geq \gamma^* x^T A$. Similarly, if $\gamma^* < \beta_0$, there is no semi-positive p so that $Bp \leq \gamma^* Ap$. Let $\gamma^* \in [\beta_0, \alpha_0]$, then $x_0^T B \geq \alpha_0 x_0^T A \geq \gamma^* x_0^T A$. Moreover, $Bp_0 \leq \beta_0 Ap_0 \leq \gamma^* Ap_0$. These two inequalities imply $x_0 \left(B - \gamma^* A\right) p_0 = 0$.

Here the constant γ^* is both expansion and 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 compactly captures the requirements that if any good grows at a rate larger than γ^* (i.e., if it is *oversupplied*), then its price must be zero; and that if any activity provides negative profit, it must be unused.

Therefore, these expressions encode all equilibrium conditions and Theorem I essentially states that under Assumptions I and II there always exists an equilibrium (γ^*, x_0, p_0) with balanced growth.

Note that Theorem I is silent about uniqueness of the equilibrium. In fact, it does not rule out (trivial) cases with $x_0^T B p_0 = 0$ so that nothing of value is produced.

To exclude such uninteresting cases, Kemeny, Morgenstern and Thomspson [96] add an extra requirement

$$x_0^T B p_0 > 0$$

and call the resulting equilibria *economic solutions*.

They show that this extra condition does not affect the existence result, while it significantly reduces the number of (relevant) solutions.

86.6 Interpretation as a Game Theoretic Problem (Two-player Zero-sum Game)

To compute the equilibrium (γ^*, x_0, p_0), we follow the algorithm proposed by Hamburger, Thompson and Weil (1967), building on the key insight that the equilibrium (with balanced growth) can be considered as a solution of a particular two-player zero-sum game. First, we introduce some notations.

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: - row player chooses the m-vector $x > \mathbf{0}$, s.t. $\iota_m^T x = 1$ - column player chooses the n-vector $p > \mathbf{0}$, s.t. $\iota_n^T p = 1$.

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)

From the famous theorem of Nash (1951), it follows that there always exists a mixed strategy Nash equilibrium for any *finite* two-player zero-sum game.

Moreover, von Neumann's Minmax Theorem [160] implies that

$$V(C) = \max_x \min_p \; x^T C p = \min_p \max_x \; x^T C p = (x^*)^T C p^*$$

86.6.1 Connection with Linear Programming (LP)

Finding Nash equilibria of a finite two-player zero-sum game can be formulated as 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: $v \equiv \min_p x^T Cp = \min_j x^T Ce^j$ - For a fixed p, let u be the value of the maximization problem: $u \equiv \max_x x^T Cp = \max_i (e^i)^T Cp$.

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

$$\begin{aligned} V(C) = & \min \quad u \\ \text{s.t.} \quad & u\iota_m \geq Cp \\ & p \geq \mathbf{0} \\ & \iota_m^T p = 1 \end{aligned}$$

Hamburger, Thompson and Weil [69] 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

$$V(-A) < 0 \qquad \text{and} \qquad V(B) > 0$$

> *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 $x^T Bp > 0$, that is 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 the matrix for $\gamma \in \mathbb{R}$

$$M(\gamma) \equiv B - \gamma A$$

For fixed γ, treating $M(\gamma)$ as a matrix game, we can calculate the solution of the game

- 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 $V(M(\beta_0)) = V(M(\alpha_0)) = 0$.

- 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 $\gamma'' < \gamma$

$$M(\gamma)p'' = M(\gamma'') + (\gamma'' - \gamma)Ap'' \leq \mathbf{0}$$

hence $V(M(\gamma)) \leq 0$.

It is clear from the above argument that β_0, α_0 are the minimal and maximal γ for which $V(M(\gamma)) = 0$.

Moreover, Hamburger et al. [69] show that the function $\gamma \mapsto V(M(\gamma))$ is continuous and non-increasing in γ.

This suggests an algorithm to compute (α_0, x_0) and (β_0, p_0) for a given input-output pair (A, B).

86.6.2 Algorithm

Hamburger, Thompson and Weil [69] propose a simple bisection algorithm to find the minimal and maximal roots (i.e. β_0 and α_0) of the function $\gamma \mapsto V(M(\gamma))$.

Step 1

First, notice that we can easily find trivial upper and lower bounds for α_0 and β_0.

- TEP requires that $x^T(B - \alpha A) \geq \mathbf{0}^T$ and $x > \mathbf{0}$, so if α is so large that $\max_i\{[(B - \alpha A)\iota_n]_i\} < 0$, then TEP ceases to have a solution.

Accordingly, let **UB** be the α^* that solves $\max_i\{[(B - \alpha^* A)\iota_n]_i\} = 0$.

- Similar to the upper bound, if β is so low that $\min_j\{[\iota_m^T(B - \beta A)]_j\} > 0$, then the EEP has no solution and so we can define **LB** as the β^* that solves $\min_j\{[\iota_m^T(B - \beta^* A)]_j\} = 0$.

The *bounds* method calculates these trivial bounds for us

```
In [6]: n1.bounds()
```

```
Out[6]: (1.0, 2.0)
```

Step 2

Compute α_0 and β_0

- Finding α_0
  1. Fix $\gamma = \frac{UB+LB}{2}$ and compute the solution of the two-player zero-sum game associated with $M(\gamma)$. We can use either the primal or the dual LP problem.
  2. If $V(M(\gamma)) \geq 0$, then set $LB = \gamma$, otherwise let $UB = \gamma$.
  3. Iterate on 1. and 2. until $|UB - LB| < \epsilon$.
- Finding β_0
  1. Fix $\gamma = \frac{UB+LB}{2}$ and compute the solution of the two-player zero-sum game associated. with $M(\gamma)$. We can use either the primal or the dual LP problem.
  2. If $V(M(\gamma)) > 0$, then set $LB = \gamma$, otherwise let $UB = \gamma$.
  3. Iterate on 1. and 2. until $|UB - LB| < \epsilon$.

  Existence: Since $V(M(LB)) > 0$ and $V(M(UB)) < 0$ and $V(M(\cdot))$ is a continuous, nonincreasing function, there is at least one $\gamma \in [LB, UB]$, s.t. $V(M(\gamma)) = 0$.

The *zerosum* method calculates the value and optimal strategies associated with a given γ.

In [7]:
```
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]
```

```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:158:
OptimizeWarning: Unknown solver options: bland
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:176:
OptimizeWarning: Unknown solver options: bland
```

In [8]:
```
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()
```

```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:158:
OptimizeWarning: Unknown solver options: bland
```

The *expansion* method implements the bisection algorithm for α_0 (and uses the primal LP problem for x_0)

```
In [9]: α_0, x, p = n1.expansion()
        print(f'α_0 = {α_0}')
        print(f'x_0 = {x}')
        print(f'The corresponding p from the dual = {p}')
```

```
α_0 = 1.2599210478365421
x_0 = [0.33 0.26 0.41]
The corresponding p from the dual = [4.13e-01 3.27e-01 2.60e-01 1.82e-10]
```

```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:158:
OptimizeWarning: Unknown solver options: bland
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:176:
OptimizeWarning: Unknown solver options: bland
```

The *interest* method implements the bisection algorithm for β_0 (and uses the dual LP problem for p_0)

```
In [10]: 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}')
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:176:
OptimizeWarning: Unknown solver options: bland
```

```
    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]
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:158:
OptimizeWarning: Unknown solver options: bland
```

Of course, when γ^* is unique, it is irrelevant which one of the two methods we use.

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. $\alpha_0 = \beta_0 = \gamma^*$, and that the expansion (and interest) rate γ^* is unique.

86.6.3 Uniqueness and Irreducibility

As an illustration, compute first the maximal and minimal roots of $V(M(\cdot))$ for Example 2, which displays a reducible input-output pair (A, B)

```
In [11]: 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}')
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:158:
OptimizeWarning: Unknown solver options: bland
```

```
    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]
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:176:
OptimizeWarning: Unknown solver options: bland
```

```
In [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}')
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:176:
OptimizeWarning: Unknown solver options: bland
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:158:
OptimizeWarning: Unknown solver options: bland
```

```
    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 γ^* that make the value of the game with $M(\gamma^*)$ zero. (see the figure above).

Indeed, although the von Neumann theorem assures existence of the equilibrium, Assumptions I and II are not sufficient for uniqueness. Nonetheless, Kemeny et al. (1967) show that there are at most finitely many economic solutions, meaning that there are only finitely many γ^* that satisfy $V(M(\gamma^*)) = 0$ and $x_0^T B p_0 > 0$ and that for each such γ_i^*, there is a self-sufficient part of the economy (a sub-economy) that in equilibrium can expand independently with the expansion coefficient γ_i^*.

The following theorem (see Theorem 9.10. in Gale [60]) asserts that imposing irreducibility is sufficient for uniqueness of (γ^*, x_0, p_0).

Theorem II: Consider the conditions of Theorem 1. If the economy (A, B) is irreducible, then $\gamma^* = \alpha_0 = \beta_0$.

86.6.4 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 1. $n = m$ 2. Each activity produces exactly one good 3. 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 [60]): 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 classical result of **Perron and Frobenius** implies that a non-negative matrix always has a non-negative eigenvalue-eigenvector pair.

Moreover, if A is irreducible, then the optimal intensity vector x_0 is positive and *unique* up to multiplication by a positive scalar.

Suppose that A is reducible with k irreducible subsets $S_1, \dots, S_k$. Let A_i be the submatrix corresponding to S_i and let α_i and β_i be the associated expansion and interest factors, respectively. Then we have

$$\alpha_0 = \max_i \{\alpha_i\} \qquad \text{and} \qquad \beta_0 = \min_i \{\beta_i\}$$

Part XII

Time Series Models

Chapter 87

Covariance Stationary Processes

87.1 Contents

- Overview 87.2
- Introduction 87.3
- Spectral Analysis 87.4
- Implementation 87.5

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

87.2 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.

87.2.1 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.

87.2.2 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.

87.2.3 Other Reading

For supplementary reading, see

- [108], chapter 2
- [142], chapter 11
- John Cochrane's notes on time series analysis, chapter 8
- [147], chapter 6
- [38], all

Let's start with some imports:

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import quantecon as qe
```

87.3 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).

87.3.1 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.

87.3.2 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.

87.3.3 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*

$$X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}, \qquad t \in \mathbb{Z} \tag{1}$$

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 (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 (1) is

$$\gamma(k) = \sigma^2 \sum_{j=0}^{\infty} \psi_j \psi_{j+k} \tag{2}$$

By the Cauchy-Schwartz inequality, one can show that $\gamma(k)$ satisfies equation (2).

Evidently, $\gamma(k)$ does not depend on t.

87.3.4 Wold Representation

Remarkably, the class of general linear processes goes a long way towards describing the entire class of zero-mean covariance stationary processes.

In particular, Wold's decomposition theorem states that every zero-mean covariance stationary process $\{X_t\}$ can be written as

$$X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j} + \eta_t$$

where

- $\{\epsilon_t\}$ is white noise
- $\{\psi_t\}$ is square summable
- $\psi_0 \epsilon_t$ is the one-step ahead prediction error in forecasting X_t as a linear least-squares function of the infinite history $X_{t-1}, X_{t-2}, \ldots$
- η_t can be expressed as a linear function of $X_{t-1}, X_{t-2}, \ldots$ and is perfectly predictable over arbitrarily long horizons

For the method of constructing a Wold representation, intuition, and further discussion, see [142], p. 286.

87.3.5 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

$$X_t = \phi X_{t-1} + \epsilon_t \quad \text{where} \quad |\phi| < 1 \quad \text{and } \{\epsilon_t\} \text{ is white noise} \tag{3}$$

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 (2) to the previous expression for X_t, we get the AR(1) autocovariance function

$$\gamma(k) = \phi^k \frac{\sigma^2}{1-\phi^2}, \qquad k = 0, 1, \ldots \tag{4}$$

The next figure plots an example of this function for $\phi = 0.8$ and $\phi = -0.8$ with $\sigma = 1$.

In [3]:
```
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')
    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

87.3.6 ARMA Processes

A stochastic process $\{X_t\}$ is called an *autoregressive moving average process*, or ARMA(p, q), if it can be written as

$$X_t = \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q} \tag{5}$$

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 (5) as

$$L^0 X_t - \phi_1 L^1 X_t - \cdots - \phi_p L^p X_t = L^0 \epsilon_t + \theta_1 L^1 \epsilon_t + \cdots + \theta_q L^q \epsilon_t \tag{6}$$

If we let $\phi(z)$ and $\theta(z)$ be the polynomials

$$\phi(z) := 1 - \phi_1 z - \cdots - \phi_p z^p \quad \text{and} \quad \theta(z) := 1 + \theta_1 z + \cdots + \theta_q z^q \tag{7}$$

then (6) becomes

$$\phi(L) X_t = \theta(L) \epsilon_t \tag{8}$$

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 [38].

The function $t \mapsto \psi_t$ is often called the *impulse response function*.

87.4 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.

87.4.1 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 $\tan(\omega) = y/x$.

87.4.2 Spectral Densities

Let $\{X_t\}$ be a covariance stationary process with autocovariance function γ satisfying $\sum_k \gamma(k)^2 < \infty$.

The *spectral density* f of $\{X_t\}$ is defined as the discrete time Fourier transform of its autocovariance function γ.

$$f(\omega) := \sum_{k \in \mathbb{Z}} \gamma(k) e^{-i\omega k}, \qquad \omega \in \mathbb{R}$$

(Some authors normalize the expression on the right by constants such as $1/\pi$ — the convention chosen makes little difference provided you are consistent).

Using the fact that γ is *even*, in the sense that $\gamma(t) = \gamma(-t)$ for all t, we can show that

$$f(\omega) = \gamma(0) + 2\sum_{k \geq 1} \gamma(k)\cos(\omega k) \tag{9}$$

It is not difficult to confirm that f is

- real-valued
- even ($f(\omega) = f(-\omega)$), and
- 2π-periodic, in the sense that $f(2\pi + \omega) = f(\omega)$ for all ω

It follows that the values of f on $[0, \pi]$ determine the values of f on all of $\mathbb{R}$ — the proof is an exercise.

For this reason, it is standard to plot the spectral density only on the interval $[0, \pi]$.

87.4.3 Example 1: White Noise

Consider a white noise process $\{\epsilon_t\}$ with standard deviation σ.

It is easy to check that in this case $f(\omega) = \sigma^2$. So f is a constant function.

As we will see, this can be interpreted as meaning that "all frequencies are equally present".

(White light has this property when frequency refers to the visible spectrum, a connection that provides the origins of the term "white noise")

87.4.4 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

$$f(\omega) = \sigma^2(1 + 2\theta\cos(\omega) + \theta^2) \tag{10}$$

With a bit more effort, it's possible to show (see, e.g., p. 261 of [142]) that the spectral density of the AR(1) process $X_t = \phi X_{t-1} + \epsilon_t$ is

$$f(\omega) = \frac{\sigma^2}{1 - 2\phi\cos(\omega) + \phi^2} \tag{11}$$

More generally, it can be shown that the spectral density of the ARMA process (5) is

$$f(\omega) = \left|\frac{\theta(e^{i\omega})}{\phi(e^{i\omega})}\right|^2 \sigma^2 \tag{12}$$

where

- σ is the standard deviation of the white noise process $\{\epsilon_t\}$.
- the polynomials $\phi(\cdot)$ and $\theta(\cdot)$ are as defined in (7).

The derivation of (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, [142], chapter 11, section 4.

It's a nice exercise to verify that (10) and (11) are indeed special cases of (12).

87.4.5 Interpreting the Spectral Density

Plotting (11) reveals the shape of the spectral density for the AR(1) model when ϕ takes the values 0.8 and -0.8 respectively.

```
In [4]: def ar1_sd(ϕ, ω):
            return 1 / (1 - 2 * ϕ * np.cos(ω) + ϕ**2)

        ωs = np.linspace(0, np.pi, 180)
        num_rows, num_cols = 2, 1
        fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 8))
        plt.subplots_adjust(hspace=0.4)

        # Autocovariance when phi = 0.8
        for i, ϕ in enumerate((0.8, -0.8)):
            ax = axes[i]
            sd = ar1_sd(ϕ, ωs)
            ax.plot(ωs, sd, 'b-', alpha=0.6, lw=2,
                    label='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

$$f(\omega) = \gamma(0) + 2\sum_{k\geq 1}\gamma(k)\cos(\omega k) = \gamma(0) + 2\sum_{k\geq 1}(-0.8)^k\cos(\omega k) \tag{13}$$

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 (13) is large.

These ideas are illustrated in the next figure, which has k on the horizontal axis.

In [5]:
```
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 ⊠ = -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()
```

```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:27:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a
LineCollection instead of individual lines. This significantly improves the
performance of a stem plot. To remove this warning and switch to the new behaviour,
set the "use_line_collection" keyword argument to True.
```

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.

In [6]:
```
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')
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()
```

```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:27:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a
LineCollection instead of individual lines. This significantly improves the
performance of a stem plot. To remove this warning and switch to the new behaviour,
set the "use_line_collection" keyword argument to True.
```

In summary, the spectral density is large at frequencies ω where the autocovariance function exhibits damped cycles.

87.4.6 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

$$\gamma(k) = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(\omega) e^{i\omega k} d\omega \tag{14}$$

This is convenient in situations where the spectral density is easier to calculate and manipulate than the autocovariance function.

(For example, the expression (12) for the ARMA spectral density is much easier to work with than the expression for the ARMA autocovariance)

87.4.7 Mathematical Theory

This section is loosely based on [142], 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

$$f = \sum_k \alpha_k h_k \quad \text{where} \quad \alpha_k := \langle f, h_k \rangle \tag{15}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product in H.

Thus, f can be represented to any degree of precision by linearly combining basis vectors.

The scalar sequence $\alpha = \{\alpha_k\}$ is called the *Fourier coefficients* of f, and satisfies $\sum_k |\alpha_k|^2 < \infty$.

In other words, α is in ℓ_2, the set of square summable sequences.

Consider an operator T that maps $\alpha \in \ell_2$ into its expansion $\sum_k \alpha_k h_k \in H$.

The Fourier coefficients of $T\alpha$ are just $\alpha = \{\alpha_k\}$, as you can verify by confirming that $\langle T\alpha, h_k \rangle = \alpha_k$.

Using elementary results from Hilbert space theory, it can be shown that

- T is one-to-one — if α and β are distinct in ℓ_2, then so are their expansions in H.
- T is onto — if $f \in H$ then its preimage in ℓ_2 is the sequence α given by $\alpha_k = \langle f, h_k \rangle$.
- T is a linear isometry — in particular, $\langle \alpha, \beta \rangle = \langle T\alpha, T\beta \rangle$.

Summarizing these results, we say that any separable Hilbert space is isometrically isomorphic to ℓ_2.

In essence, this says that each separable Hilbert space we consider is just a different way of looking at the fundamental space ℓ_2.

With this in mind, let's specialize to a setting where

- $\gamma \in \ell_2$ is the autocovariance function of a covariance stationary process, and f is the spectral density.
- $H = L_2$, where L_2 is the set of square summable functions on the interval $[-\pi, \pi]$, with inner product $\langle g, h \rangle = \int_{-\pi}^{\pi} g(\omega)h(\omega)d\omega$.
- $\{h_k\}$ = the orthonormal basis for L_2 given by the set of trigonometric functions.

$$h_k(\omega) = \frac{e^{i\omega k}}{\sqrt{2\pi}}, \quad k \in \mathbb{Z}, \quad \omega \in [-\pi, \pi]$$

Using the definition of T from above and the fact that f is even, we now have

$$T\gamma = \sum_{k \in \mathbb{Z}} \gamma(k) \frac{e^{i\omega k}}{\sqrt{2\pi}} = \frac{1}{\sqrt{2\pi}} f(\omega) \tag{16}$$

In other words, apart from a scalar multiple, the spectral density is just a transformation of $\gamma \in \ell_2$ under a certain linear isometry — a different way to view γ.

In particular, it is an expansion of the autocovariance function with respect to the trigonometric basis functions in L_2.

As discussed above, the Fourier coefficients of $T\gamma$ are given by the sequence γ, and, in particular, $\gamma(k) = \langle T\gamma, h_k \rangle$.

Transforming this inner product into its integral expression and using (16) gives (14), justifying our earlier expression for the inverse transform.

87.5 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

87.5.1 Application

Let's use this code to replicate the plots on pages 68–69 of [108].

Here are some functions to generate the plots

```
In [7]: 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):
    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=(12, 8))
    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 $X_t = \epsilon_t$.

```
In [8]: ϕ = 0.0
        θ = 0.0
```

```
arma = qe.ARMA(ϕ, θ)
quad_plot(arma)
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:5:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a
LineCollection instead of individual lines. This significantly improves the
performance of a stem plot. To remove this warning and switch to the new behaviour,
set the "use_line_collection" keyword argument to True.
  """
/home/ubuntu/anaconda3/lib/python3.7/site-packages/numpy/core/numeric.py:538:
ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:16:
UserWarning: Attempted to set non-positive bottom ylim on a log-scaled axis.
Invalid limit will be ignored.
  app.launch_new_instance()
/home/ubuntu/anaconda3/lib/python3.7/site-packages/matplotlib/transforms.py:923:
ComplexWarning: Casting complex values to real discards the imaginary part
  self._points[:, 1] = interval
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:23:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a
LineCollection instead of individual lines. This significantly improves the
performance of a stem plot. To remove this warning and switch to the new behaviour,
set the "use_line_collection" keyword argument to True.
```

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 [108] use to teach "how to read spectral densities".

Ljunqvist and Sargent's first model is $X_t = 1.3X_{t-1} - .7X_{t-2} + \epsilon_t$

```
In [9]: ϕ = 1.3, -.7
        θ = 0.0
        arma = qe.ARMA(ϕ, θ)
        quad_plot(arma)
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:5:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a
LineCollection instead of individual lines. This significantly improves the
performance of a stem plot. To remove this warning and switch to the new behaviour,
set the "use_line_collection" keyword argument to True.
  """
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:16:
UserWarning: Attempted to set non-positive bottom ylim on a log-scaled axis.
Invalid limit will be ignored.
  app.launch_new_instance()
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:23:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a
LineCollection instead of individual lines. This significantly improves the
performance of a stem plot. To remove this warning and switch to the new behaviour,
set the "use_line_collection" keyword argument to True.
```

Ljungqvist and Sargent's second model is $X_t = .9X_{t-1} + \epsilon_t$

```
In [10]: ϕ = 0.9
         θ = -0.0
```

```
arma = qe.ARMA(ϕ, θ)
quad_plot(arma)
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:5:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a
LineCollection instead of individual lines. This significantly improves the
performance of a stem plot. To remove this warning and switch to the new behaviour,
set the "use_line_collection" keyword argument to True.
  """
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:16:
UserWarning: Attempted to set non-positive bottom ylim on a log-scaled axis.
Invalid limit will be ignored.
  app.launch_new_instance()
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:23:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a
LineCollection instead of individual lines. This significantly improves the
performance of a stem plot. To remove this warning and switch to the new behaviour,
set the "use_line_collection" keyword argument to True.
```

Ljungqvist and Sargent's third model is $X_t = .8X_{t-4} + \epsilon_t$

```
In [11]: ϕ = 0., 0., 0., .8
         θ = -0.0
         arma = qe.ARMA(ϕ, θ)
         quad_plot(arma)
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:5:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a
LineCollection instead of individual lines. This significantly improves the
performance of a stem plot. To remove this warning and switch to the new behaviour,
set the "use_line_collection" keyword argument to True.
```

```
  """
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:16:
UserWarning: Attempted to set non-positive bottom ylim on a log-scaled axis.
Invalid limit will be ignored.
  app.launch_new_instance()
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:23:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a
LineCollection instead of individual lines. This significantly improves the
performance of a stem plot. To remove this warning and switch to the new behaviour,
set the "use_line_collection" keyword argument to True.
```

Ljungqvist and Sargent's fourth model is $X_t = .98X_{t-1} + \epsilon_t - .7\epsilon_{t-1}$

```
In [12]: ϕ = .98
         θ = -0.7
         arma = qe.ARMA(ϕ, θ)
         quad_plot(arma)
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:5:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a
LineCollection instead of individual lines. This significantly improves the
performance of a stem plot. To remove this warning and switch to the new behaviour,
set the "use_line_collection" keyword argument to True.
  """
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:16:
UserWarning: Attempted to set non-positive bottom ylim on a log-scaled axis.
Invalid limit will be ignored.
  app.launch_new_instance()
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:23:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a
LineCollection instead of individual lines. This significantly improves the
performance of a stem plot. To remove this warning and switch to the new behaviour,
```

```
set the "use_line_collection" keyword argument to True.
```

87.5.2 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 (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 (7) — that is,

- `ma_poly` is the vector $(1, \theta_1, \theta_2, \dots, \theta_q)$
- `ar_poly` is the vector $(1, -\phi_1, -\phi_2, \dots, -\phi_p)$

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.

87.5.3 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 $A_0, A_1, \dots, A_{n-1}$ and returns the sequence $a_0, a_1, \dots, a_{n-1}$ 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 (14), we have now shown that, for n sufficiently large, $a_k \approx \gamma(k)$ — which is exactly what we want to compute.

Chapter 88

Estimation of Spectra

88.1 Contents

- Overview 88.2
- Periodograms 88.3
- Smoothing 88.4
- Exercises 88.5
- Solutions 88.6

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

88.2 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 [142] or [38].

Let's start with some standard imports:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        from quantecon import ARMA, periodogram, ar_periodogram
```

88.3 Periodograms

Recall that the spectral density f of a covariance stationary process with autocorrelation function γ can be written

$$f(\omega) = \gamma(0) + 2\sum_{k\geq 1}\gamma(k)\cos(\omega k), \qquad \omega \in \mathbb{R}$$

Now consider the problem of estimating the spectral density of a given time series, when γ is unknown.

In particular, let $X_0, \dots, X_{n-1}$ 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

$$I(\omega) := \frac{1}{n}\left|\sum_{t=0}^{n-1} X_t e^{it\omega}\right|^2, \qquad \omega \in \mathbb{R} \tag{1}$$

(Recall that $|z|$ denotes the modulus of complex number z)

Alternatively, $I(\omega)$ can be expressed as

$$I(\omega) = \frac{1}{n}\left\{\left[\sum_{t=0}^{n-1} X_t\cos(\omega t)\right]^2 + \left[\sum_{t=0}^{n-1} X_t\sin(\omega t)\right]^2\right\}$$

It is straightforward to show that the function I is even and 2π-periodic (i.e., $I(\omega) = I(-\omega)$ and $I(\omega + 2\pi) = I(\omega)$ for all $\omega \in \mathbb{R}$).

From these two results, you will be able to verify that the values of I on $[0, \pi]$ determine the values of I on all of $\mathbb{R}$.

The next section helps to explain the connection between the periodogram and the spectral density.

88.3.1 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 γ.

("Plug-in estimator" is an informal term for an estimator found by replacing expectations with sample means)

With this notation, we can now write

$$I(\omega_j) = \hat{\gamma}(0) + 2\sum_{k=1}^{n-1}\hat{\gamma}(k)\cos(\omega_j k)$$

Recalling our expression for f given above, we see that $I(\omega_j)$ is just a sample analog of $f(\omega_j)$.

88.3.2 Calculation

Let's now consider how to compute the periodogram as defined in (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

$$A_j := \sum_{t=0}^{n-1} a_t \exp\left\{i2\pi\frac{tj}{n}\right\}, \qquad j = 0, \dots, n-1$$

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

$$\frac{1}{n}\left|\sum_{t=0}^{n-1} X_t \exp\left\{i2\pi\frac{tj}{n}\right\}\right|^2, \qquad j = 0, \dots, n-1$$

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

$$X_t = 0.5X_{t-1} + \epsilon_t - 0.8\epsilon_{t-2} \tag{2}$$

where $\{\epsilon_t\}$ is white noise with unit variance, and compares the periodogram to the actual spectral density

In [3]:
```
n = 40                          # Data size
ϕ, θ = 0.5, (0, -0.8)           # AR and MA parameters
lp = ARMA(ϕ, θ)
X = lp.simulation(ts_length=n)

fig, ax = plt.subplots()
x, y = periodogram(X)
ax.plot(x, y, 'b-', lw=2, alpha=0.5, label='periodogram')
x_sd, y_sd = lp.spectral_density(two_pi=False, res=120)
ax.plot(x_sd, y_sd, 'r-', lw=2, alpha=0.8, label='spectral density')
ax.legend()
plt.show()
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/numpy/core/numeric.py:538:
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.

88.4 Smoothing

There are two related issues here.

One is that, given the way the fast Fourier transform is implemented, the number of points ω at which $I(\omega)$ is estimated increases in line with the amount of data.

In other words, although we have more data, we are also using it to estimate more values.

A second issue is that densities of all types are fundamentally hard to estimate without parametric assumptions.

Typically, nonparametric estimation of densities requires some degree of smoothing.

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

$$I_S(\omega_j) := \sum_{\ell=-p}^{p} w(\ell) I(\omega_{j+\ell}) \tag{3}$$

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 (3).

In [4]:
```
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()
```

88.4.1 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 (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.

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.

88.4.2 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 (3).

The underlying assumption that makes this a good idea is that the true spectral density is relatively regular — the value of $I(\omega)$ is close to that of $I(\omega')$ when ω is close to ω'.

This will not be true in all cases, but it is certainly true for white noise.

For white noise, I is as regular as possible — it is a constant function.

In this case, values of $I(\omega')$ at points ω' near to ω provided the maximum possible amount of information about the value $I(\omega)$.

Another way to put this is that if I is relatively constant, then we can use a large amount of smoothing without introducing too much bias.

88.4.3 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

$$X_{t+1} = \mu + \phi X_t + \epsilon_{t+1} \tag{4}$$

where μ and $\phi \in (-1, 1)$ are unknown parameters and $\{\epsilon_t\}$ is white noise.

It follows that if we regress X_{t+1} on X_t and an intercept, the residuals will approximate white noise.

Let

- g be the spectral density of $\{\epsilon_t\}$ — a constant function, as discussed above
- I_0 be the periodogram estimated from the residuals — an estimate of g
- f be the spectral density of $\{X_t\}$ — the object we are trying to estimate

In view of an earlier result we obtained while discussing ARMA processes, f and g are related by

$$f(\omega) = \left| \frac{1}{1 - \phi e^{i\omega}} \right|^2 g(\omega) \tag{5}$$

This suggests that the recoloring step, which constructs an estimate I of f from I_0, should set

$$I(\omega) = \left| \frac{1}{1 - \hat{\phi} e^{i\omega}} \right|^2 I_0(\omega)$$

where $\hat{\phi}$ is the OLS estimate of ϕ.

The code for `ar_periodogram()` — the third function in `estspec.py` — does exactly this. (See the code here).

The next figure shows realizations of the two kinds of smoothed periodograms

1. "standard smoothed periodogram", the ordinary smoothed periodogram, and

2. "AR smoothed periodogram", the pre-whitened and recolored one generated by `ar_periodogram()`

The periodograms are calculated from time series drawn from (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.

88.5 Exercises

88.5.1 Exercise 1

Replicate this figure (modulo randomness).

The model is as in equation (2) and there are 400 observations.

For the smoothed periodogram, the window type is "hamming".

88.5.2 Exercise 2

Replicate this figure (modulo randomness).

The model is as in equation (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.

88.6 Solutions

88.6.1 Exercise 1

In [5]:
```
## 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()
```

88.6.2 Exercise 2

```
In [6]: 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()
```

Chapter 89

Additive and Multiplicative Functionals

89.1 Contents

- Overview 89.2
- A Particular Additive Functional 89.3
- Dynamics 89.4
- Code 89.5
- More About the Multiplicative Martingale 89.6

Co-authors: Chase Coleman and Balint Szoke

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

89.2 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 having persistent growth that still enables statistical learning based on a law of large number for an asymptotically stationary and ergodic process?

The answer provided by Hansen and Scheinkman [79] 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 [77] (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 [77].

Let's start with some imports:

In [2]:
```
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
```

89.3 A Particular Additive Functional

Hansen and Sargent [77] 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)

$$x_{t+1} = Ax_t + Bz_{t+1} \tag{1}$$

Here

- x_t is an $n \times 1$ vector,
- A is an $n \times n$ stable matrix (all eigenvalues lie within the open unit circle),
- $z_{t+1} \sim N(0, I)$ is an $m \times 1$ IID shock,
- B is an $n \times m$ matrix, and

- $x_0 \sim N(\mu_0, \Sigma_0)$ is a random initial condition for x

The second piece is an equation that expresses increments of $\{y_t\}_{t=0}^{\infty}$ as linear functions of

- a scalar constant ν,
- the vector x_t, and
- the same Gaussian vector z_{t+1} that appears in the VAR (1)

In particular,

$$y_{t+1} - y_t = \nu + Dx_t + Fz_{t+1} \tag{2}$$

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*.

89.3.1 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.

89.4 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.

$$\tilde{x}_{t+1} = \phi_1 \tilde{x}_t + \phi_2 \tilde{x}_{t-1} + \phi_3 \tilde{x}_{t-2} + \phi_4 \tilde{x}_{t-3} + \sigma z_{t+1} \tag{3}$$

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 (3) is not a first order system like (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 (1) – (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.

89.4.1 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 [79].

All of these objects are computed using the code below

In [3]:
```
"""
@authors: Chase Coleman, Balint Szoke, Tom Sargent

"""

class AMF_LSS_VAR:
    """
    This class transforms an additive (multiplicative)
    functional into a QuantEcon linear state space system.
    """

    def __init__(self, A, B, D, F=None, ν=None):
        # Unpack required elements
        self.nx, self.nk = B.shape
        self.A, self.B = A, B

        # Checking the dimension of D (extended from the scalar case)
        if len(D.shape) > 1 and D.shape[0] != 1:
            self.nm = D.shape[0]
```

```
            self.D = D
        elif len(D.shape) > 1 and D.shape[0] == 1:
            self.nm = 1
            self.D = D
        else:
            self.nm = 1
            self.D = np.expand_dims(D, 0)

        # Create space for additive decomposition
        self.add_decomp = None
        self.mult_decomp = None

        # Set F
        if not np.any(F):
            self.F = np.zeros((self.nk, 1))
        else:
            self.F = F

        # Set ν
        if not np.any(ν):
            self.ν = np.zeros((self.nm, 1))
        elif type(ν) == float:
            self.ν = np.asarray([[ν]])
        elif len(ν.shape) == 1:
            self.ν = np.expand_dims(ν, 1)
        else:
            self.ν = ν

        if self.ν.shape[0] != self.D.shape[0]:
            raise ValueError("The dimension of ν is inconsistent with D!")

        # Construct BIG state space representation
        self.lss = self.construct_ss()

    def construct_ss(self):
        """
        This creates the state space representation that can be passed
        into the quantecon LSS class.
        """
        # Pull out useful info
        nx, nk, nm = self.nx, self.nk, self.nm
        A, B, D, F, ν = self.A, self.B, self.D, self.F, self.ν
        if self.add_decomp:
            ν, H, g = self.add_decomp
        else:
            ν, H, g = self.additive_decomp()

        # Auxiliary blocks with 0's and 1's to fill out the lss matrices
        nx0c = np.zeros((nx, 1))
        nx0r = np.zeros(nx)
        nx1 = np.ones(nx)
        nk0 = np.zeros(nk)
        ny0c = np.zeros((nm, 1))
        ny0r = np.zeros(nm)
        ny1m = np.eye(nm)
        ny0m = np.zeros((nm, nm))
        nyx0m = np.zeros_like(D)
```

```
        # Build A matrix for LSS
        # Order of states is: [1, t, xt, yt, mt]
        A1 = np.hstack([1, 0, nx0r, ny0r, ny0r])         # Transition for 1
        A2 = np.hstack([1, 1, nx0r, ny0r, ny0r])         # Transition for t
        # Transition for x_{t+1}
        A3 = np.hstack([nx0c, nx0c, A, nyx0m.T, nyx0m.T])
        # Transition for y_{t+1}
        A4 = np.hstack([ν, ny0c, D, ny1m, ny0m])
        # Transition for m_{t+1}
        A5 = np.hstack([ny0c, ny0c, nyx0m, ny0m, ny1m])
        Abar = np.vstack([A1, A2, A3, A4, A5])

        # Build B matrix for LSS
        Bbar = np.vstack([nk0, nk0, B, F, H])

        # Build G matrix for LSS
        # Order of observation is: [xt, yt, mt, st, tt]
        # Selector for x_{t}
        G1 = np.hstack([nx0c, nx0c, np.eye(nx), nyx0m.T, nyx0m.T])
        G2 = np.hstack([ny0c, ny0c, nyx0m, ny1m, ny0m])   # Selector for
↪y_{t}
        # Selector for martingale
        G3 = np.hstack([ny0c, ny0c, nyx0m, ny0m, ny1m])
        G4 = np.hstack([ny0c, ny0c, -g, ny0m, ny0m])  # Selector for
↪stationary
        G5 = np.hstack([ny0c, ν, nyx0m, ny0m, ny0m])  # Selector for trend
        Gbar = np.vstack([G1, G2, G3, G4, G5])

        # Build H matrix for LSS
        Hbar = np.zeros((Gbar.shape[0], nk))

        # Build LSS type
        x0 = np.hstack([1, 0, nx0r, ny0r, ny0r])
        S0 = np.zeros((len(x0), len(x0)))
        lss = qe.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]


def plot_additive(self, T, npaths=25, show_trend=True):
    """
    Plots for the additive decomposition

    """
    # Pull out right sizes so we know how to increment
    nx, nk, nm = self.nx, self.nk, self.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 = self.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
            madd_dist = norm(ymeans[nx+nm+ii],
                             np.sqrt(yvar[nx+nm+ii, nx+nm+ii]))
            mbounds[li:ui, t] = madd_dist.ppf([0.01, .99])

            sadd_dist = norm(ymeans[nx+2*nm+ii],
```

```
                                np.sqrt(yvar[nx+2*nm+ii, nx+2*nm+ii]))
            sbounds[li:ui, t] = sadd_dist.ppf([0.01, .99])

        # Pull out paths
        for n in range(npaths):
            x, y = self.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(self.plot_given_paths(T,
                                                  ypath[li:ui,:],
                                                  mpath[li:ui,:],
                                                  spath[li:ui,:],
                                                  tpath[li:ui,:],
                                                  mbounds[LI:UI,:],
                                                  sbounds[LI:UI,:],
                                                  show_trend=show_trend))

            add_figs[ii].suptitle(f'Additive decomposition of $y_{ii+1}$',
                                  fontsize=14)

        return add_figs


    def plot_multiplicative(self, T, npaths=25, show_trend=True):
        """
        Plots for the multiplicative decomposition

        """
        # Pull out right sizes so we know how to increment
        nx, nk, nm = self.nx, self.nk, self.nm
        # Matrices for the multiplicative decomposition
        ν_tilde, H, g = self.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 = self.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.asscalar(np.sqrt(yvar[nx+nm+ii,
↪nx+nm+ii])),
                                scale=np.asscalar(np.exp(ymeans[nx+nm+ii] \
                                                        - t * (.5)
                                                        * np.expand_dims(
                                                          np.diag(H @ H.T),
                                                            1
                                                            )[ii]
                                                        )
                                                  )
                                )
                Sdist = lognorm(np.asscalar(np.sqrt(yvar[nx+2*nm+ii,
                                                         nx+2*nm+ii])),
                                scale = np.asscalar(
                                            np.exp(-ymeans[nx+2*nm+ii])
                                            )
                                )
                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 = self.lss.simulate(T)
            for ii in range(nm):
                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(self.plot_given_paths(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_martingales(self, T, npaths=25):

        # Pull out right sizes so we know how to increment
        nx, nk, nm = self.nx, self.nk, self.nm
        # Matrices for the multiplicative decomposition
        ν_tilde, H, g = self.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 = self.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.asscalar(np.sqrt(yvar[nx+nm+ii,
↪nx+nm+ii])),
                                scale=np.asscalar( np.exp(ymeans[nx+nm+ii] \
                                                          - t * (.5)
                                                          * np.expand_dims(
                                                          np.diag(H @ H.T),
                                                                1)[ii]
                                                         )
                                                 )
                                )
                mbounds_mult[li:ui, t] = Mdist.ppf([.01, .99])

        # Pull out paths
        for n in range(npaths):
            x, y = self.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(self.plot_martingale_paths(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


    def plot_given_paths(self, 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_martingale_paths(self, 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
```

For now, we just plot y_t and x_t, postponing until later a description of exactly how we compute them.

```
In [4]: 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.

89.4.2 Decomposition

Hansen and Sargent [77] 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 (1) and (2), we first construct the matrices

$$
\begin{aligned}
H &:= F + B'(I - A')^{-1}D \\
g &:= D'(I - A)^{-1}
\end{aligned}
$$

Then the Hansen-Scheinkman [79] 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 (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 τ_t, m_t, s_t of the decomposition.

A convenient way to do this is to construct an appropriate instance of a linear state space system by using LinearStateSpace from QuantEcon.py.

This will allow us to use the routines in LinearStateSpace to study dynamics.

To start, observe that, under the dynamics in (1) and (2) and with the definitions just given,

$$\begin{bmatrix} 1 \\ t+1 \\ x_{t+1} \\ y_{t+1} \\ m_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & A & 0 & 0 \\ \nu & 0 & D' & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ x_t \\ y_t \\ m_t \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ B \\ F' \\ H' \end{bmatrix} z_{t+1}$$

and

$$\begin{bmatrix} x_t \\ y_t \\ \tau_t \\ m_t \\ s_t \end{bmatrix} = \begin{bmatrix} 0 & 0 & I & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & \nu & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -g & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ x_t \\ y_t \\ m_t \end{bmatrix}$$

With

$$\tilde{x} := \begin{bmatrix} 1 \\ t \\ x_t \\ y_t \\ m_t \end{bmatrix} \quad \text{and} \quad \tilde{y} := \begin{bmatrix} x_t \\ y_t \\ \tau_t \\ m_t \\ s_t \end{bmatrix}$$

we can write this as the linear state space system

$$\begin{aligned} \tilde{x}_{t+1} &= \tilde{A}\tilde{x}_t + \tilde{B} z_{t+1} \\ \tilde{y}_t &= \tilde{D}\tilde{x}_t \end{aligned}$$

By picking out components of $\tilde{y}_t$, we can track all variables of interest.

89.5 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

```
In [5]: amf.plot_additive(T)
        plt.show()
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-
packages/scipy/stats/_distn_infrastructure.py:1983: RuntimeWarning: invalid value
encountered in multiply
  lower_bound = _a * scale + loc
/home/ubuntu/anaconda3/lib/python3.7/site-
packages/scipy/stats/_distn_infrastructure.py:1984: RuntimeWarning: invalid value
encountered in multiply
  upper_bound = _b * scale + loc
```

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

89.5.1 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

$$\tilde{\nu} = \nu + \frac{H \cdot H}{2}, \quad \widetilde{M}_t = \exp\Big(\sum_{j=1}^{t}\Big(H \cdot z_j - \frac{H \cdot H}{2}\Big)\Big), \quad \widetilde{M}_0 = 1$$

and

$$\tilde{e}(x) = \exp[g(x)] = \exp[D'(I-A)^{-1}x]$$

An instance of class `AMF_LSS_VAR` (above) includes this associated multiplicative functional as an attribute.

Let's plot this multiplicative functional for our example.

If you run the code that first simulated that example again and then the method call in the cell below you'll obtain the graph in the next cell.

```
In [6]: amf.plot_multiplicative(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.

89.5.2 Peculiar Large Sample Property

Hansen and Sargent [77] (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 [77].

The following simulation of many paths of $\widetilde{M}_t$ illustrates both properties

```
In [7]: np.random.seed(10021987)
        amf.plot_martingales(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.

89.6 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 $H = [F + B'(I - A')^{-1}D]$.

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.

89.6.1 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.

89.6.2 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.

In [8]:
```
"""

@authors: Chase Coleman, Balint Skoze, Tom Sargent

"""

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):
        """
        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

        return self.ν, H, g

    def multiplicative_decomp(self):
        """
        Return values for the multiplicative decomposition (Example 5.4.4.)
            - ν_tilde  : eigenvalue
            - H        : vector for the Jensen term
        """
        ν, H, g = self.additive_decomp()
        ν_tilde = ν + (.5) * H**2

        return ν_tilde, H, g

    def loglikelihood_path(self, x, y):
        A, B, D, F = self.A, self.B, self.D, self.F
        T = y.T.size
        FF = F**2
        FFinv = 1 / FF
        temp = y[1:] - y[:-1] - D * x[:-1]
        obs = temp * FFinv * temp
        obssum = np.cumsum(obs)
        scalar = (np.log(FF) + np.log(2 * np.pi)) * np.arange(1, T)

        return (-0.5) * (obssum + scalar)

    def loglikelihood(self, x, y):
```

```
        llh = self.loglikelihood_path(x, y)

        return llh[-1]
```

The heavy lifting is done inside the AMF_LSS_VAR class.

The following code adds some simple functions that make it straightforward to generate sample paths from an instance of AMF_LSS_VAR.

```
In [9]: 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
                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 took kit, let's apply them to run some simulations.

```
In [10]: 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

```
In [11]: def Mtilde_t_density(amf, t, xmin=1e-8, xmax=5.0, npts=5000):

             # Pull out the multiplicative decomposition
             ν̃tilde, 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
    ν̃, 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=(8, 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):
    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()
```

Densities of $\tilde{M}_t$
Density for time 10
Density for time 100
Density for time 500
Density for time 1000
Density for time 2500
Density for time 5000

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.

89.6.3 Multiplicative Martingale as Likelihood Ratio Process

A forthcoming lecture studies **likelihood processes** and **likelihood ratio processes**.

A **likelihood ratio process** is defined as a multiplicative martingale with mean unity.

Likelihood ratio processes exhibit the peculiar property discussed here.

We'll discuss how to interpret that property in the forthcoming lecture.

Chapter 90

Classical Control with Linear Algebra

90.1 Contents

- Overview 90.2
- A Control Problem 90.3
- Finite Horizon Theory 90.4
- The Infinite Horizon Limit 90.5
- Undiscounted Problems 90.6
- Implementation 90.7
- Exercises 90.8

90.2 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:

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

90.2.1 References

Useful references include [163], [74], [123], [11], and [120].

90.3 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 $d_0, d_1, \dots, d_m$ is a given scalar sequence.

Consider the discrete-time control problem

$$\max_{\{y_t\}} \lim_{N\to\infty} \sum_{t=0}^{N} \beta^t \left\{ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} \left[d(L) y_t\right]^2 \right\}, \tag{1}$$

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 (1) is subject to initial conditions for $y_{-1}, y_{-2} \dots, y_{-m}$.

Maximization is over infinite sequences $\{y_t\}_{t\geq 0}$.

90.3.1 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 (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 [142].

90.4 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 (1) are not sufficient for maximizing (1))

We begin by

1. fixing $N > m$,
2. differentiating the finite version of (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 $t = 0,\ 1,\ \ldots,\ N-m$ gives

$$
\begin{aligned}
\frac{\partial J}{\partial y_t} &= 2\beta^t\, d_0\, d(L)y_t + 2\beta^{t+1}\, d_1\, d(L)y_{t+1} + \cdots + 2\beta^{t+m}\, d_m\, d(L)y_{t+m} \\
&= 2\beta^t \left(d_0 + d_1\, \beta L^{-1} + d_2\, \beta^2\, L^{-2} + \cdots + d_m\, \beta^m\, L^{-m}\right) d(L)y_t
\end{aligned}
$$

We can write this more succinctly as

$$
\frac{\partial J}{\partial y_t} = 2\beta^t\, d(\beta L^{-1})\, d(L)y_t \tag{2}
$$

Differentiating J with respect to y_t for $t = N-m+1, \ldots, N$ gives

$$
\begin{aligned}
\frac{\partial J}{\partial y_N} &= 2\beta^N\, d_0\, d(L)y_N \\
\frac{\partial J}{\partial y_{N-1}} &= 2\beta^{N-1}\left[d_0 + \beta\, d_1\, L^{-1}\right] d(L)y_{N-1} \\
\vdots \qquad & \quad \vdots \\
\frac{\partial J}{\partial y_{N-m+1}} &= 2\beta^{N-m+1}\left[d_0 + \beta L^{-1}\, d_1 + \cdots + \beta^{m-1}\, L^{-m+1}\, d_{m-1}\right] d(L)y_{N-m+1}
\end{aligned} \tag{3}
$$

With these preliminaries under our belts, we are ready to differentiate (1).

Differentiating (1) with respect to y_t for $t = 0, \ldots, N-m$ gives the Euler equations

$$
\left[h + d\,(\beta L^{-1})\, d(L)\right] y_t = a_t, \quad t = 0,\ 1,\ \ldots, N-m \tag{4}
$$

The system of equations (4) forms a $2 \times m$ order linear *difference equation* that must hold for the values of t indicated.

Differentiating (1) with respect to y_t for $t = N-m+1, \ldots, N$ gives the terminal conditions

$$
\begin{aligned}
\beta^N (a_N - hy_N - d_0\, d(L)y_N) &= 0 \\
\beta^{N-1}\left(a_{N-1} - hy_{N-1} - \left(d_0 + \beta\, d_1\, L^{-1}\right) d(L)\, y_{N-1}\right) &= 0 \\
&\vdots\vdots \\
\beta^{N-m+1}\left(a_{N-m+1} - hy_{N-m+1} - (d_0 + \beta L^{-1} d_1 + \cdots\ + \beta^{m-1} L^{-m+1} d_{m-1}) d(L) y_{N-m+1}\right) &= 0
\end{aligned} \tag{5}
$$

In the finite N problem, we want simultaneously to solve (4) subject to the m initial conditions $y_{-1}, \ldots, y_{-m}$ and the m terminal conditions (5).

These conditions uniquely pin down the solution of the finite N problem.

That is, for the finite N problem, conditions (4) and (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.

90.4.1 Matrix Methods

Let's look at how linear algebra can be used to tackle and shed light on the finite horizon LQ control problem.

A Single Lag Term

Let's begin with the special case in which $m = 1$.

We want to solve the system of $N + 1$ linear equations

$$\begin{aligned} [h + d\,(\beta L^{-1})\,d\,(L)]y_t &= a_t, \quad t = 0,\ 1,\ \dots,\ N-1 \\ \beta^N [a_N - h\,y_N - d_0\,d\,(L) y_N] &= 0 \end{aligned} \tag{6}$$

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 (6) as the matrix equation

$$\begin{bmatrix} (\phi_0 - d_1^2) & \phi_1 & 0 & 0 & \dots & \dots & 0 \\ \beta\phi_1 & \phi_0 & \phi_1 & 0 & \dots & \dots & 0 \\ 0 & \beta\phi_1 & \phi_0 & \phi_1 & \dots & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \dots & \dots & \dots & \beta\phi_1 & \phi_0 & \phi_1 \\ 0 & \dots & \dots & \dots & 0 & \beta\phi_1 & \phi_0 \end{bmatrix} \begin{bmatrix} y_N \\ y_{N-1} \\ y_{N-2} \\ \vdots \\ y_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} a_N \\ a_{N-1} \\ a_{N-2} \\ \vdots \\ a_1 \\ a_0 - \phi_1 y_{-1} \end{bmatrix} \tag{7}$$

or

$$W\bar{y} = \bar{a} \tag{8}$$

Notice how we have chosen to arrange the y_t's in reverse time order.

The matrix W on the left side of (7) is "almost" a Toeplitz matrix (where each descending diagonal is constant).

There are two sources of deviation from the form of a Toeplitz matrix

1. The first element differs from the remaining diagonal elements, reflecting the terminal condition.

2. The sub-diagonal elements equal β time the super-diagonal elements.

The solution of (8) can be expressed in the form

$$\bar{y} = W^{-1}\bar{a} \tag{9}$$

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 (7) or (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 (9), we obtain

$$U\bar{y} = L^{-1}\bar{a} \tag{10}$$

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 (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, (10) has the form

$$\begin{bmatrix} 1 & U_{12} & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & U_{23} & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & U_{34} & \dots & 0 & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 & U_{N,N+1} \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} y_N \\ y_{N-1} \\ y_{N-2} \\ y_{N-3} \\ \vdots \\ y_1 \\ y_0 \end{bmatrix} =$$

$$\begin{bmatrix} L_{11}^{-1} & 0 & 0 & \dots & 0 \\ L_{21}^{-1} & L_{22}^{-1} & 0 & \dots & 0 \\ L_{31}^{-1} & L_{32}^{-1} & L_{33}^{-1} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{N,1}^{-1} & L_{N,2}^{-1} & L_{N,3}^{-1} & \dots & 0 \\ L_{N+1,1}^{-1} & L_{N+1,2}^{-1} & L_{N+1,3}^{-1} & \dots & L_{N+1\,N+1}^{-1} \end{bmatrix} \begin{bmatrix} a_N \\ a_{N-1} \\ a_{N-2} \\ \vdots \\ a_1 \\ a_0 - \phi_1 y_{-1} \end{bmatrix}$$

where L_{ij}^{-1} is the (i,j) element of L^{-1} and U_{ij} is the (i,j) element of U.

Note how the left side for a given t involves y_t and one lagged value y_{t-1} while the right side involves all future values of the forcing process $a_t, a_{t+1}, \dots, a_N$.

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 ϕ_j be the coefficients in the expansion $\phi(L) = h + d(L^{-1})d(L)$.

Then the first order conditions (4) and (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

$$U\bar{y} = L^{-1}\bar{a} \tag{11}$$

$$\sum_{j=0}^{t} U_{-t+N+1,\,-t+N+j+1}\, y_{t-j} = \sum_{j=0}^{N-t} L_{-t+N+1,\,-t+N+1-j}\, \bar{a}_{t+j}\ ,$$
$$t = 0, 1, \dots, N$$

where $L^{-1}_{t,s}$ is the element in the (t, s) position of L, and similarly for U.

The left side of equation (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 $N \to \infty$, (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.

90.5 The Infinite Horizon Limit

For the infinite horizon problem, we propose to discover first-order necessary conditions by taking the limits of (4) and (5) as $N \to \infty$.

This approach is valid, and the limits of (4) and (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 (4) and (5) are, in general, not sufficient for a maximum.

That is, the limits of (5) do not provide enough information uniquely to determine the solution of the Euler equation (4) that maximizes (1).

As we shall see below, a side condition on the path of y_t that together with (4) is sufficient for an optimum is

$$\sum_{t=0}^{\infty} \beta^t\, h y_t^2 < \infty \tag{12}$$

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 (1) than does the optimal path selected by the solution procedure below.

Consider the *characteristic equation* associated with the Euler equation

$$h + d\left(\beta z^{-1}\right) d\left(z\right) = 0 \tag{13}$$

Notice that if $\tilde{z}$ is a root of equation (13), then so is $\beta\tilde{z}^{-1}$.

Thus, the roots of (13) come in "β-reciprocal" pairs.

Assume that the roots of (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 (13) can be expressed as

$$\begin{aligned} h + d(\beta z^{-1})d(z) &= z^{-m} z_0 (z - z_1) \cdots (z - z_m)(z - z_{m+1}) \cdots (z - z_{2m}) \\ &= z^{-m} z_0 (z - z_1)(z - z_2) \cdots (z - z_m)(z - \beta z_m^{-1}) \cdots (z - \beta z_2^{-1})(z - \beta z_1^{-1}) \end{aligned} \tag{14}$$

where z_0 is a constant.

In (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

$$c\left(z\right) = \left[(-1)^m z_0 \, z_1 \cdots z_m\right]^{1/2} (1 - \frac{z}{z_1}) \, (1 - \frac{z}{z_2}) \cdots (1 - \frac{z}{z_m}) \tag{15}$$

Notice that (14) can be written

$$h + d\ \left(\beta z^{-1}\right)\ d\ \left(z\right) = c\left(\beta z^{-1}\right) c\left(z\right) \tag{16}$$

It is useful to write (15) as

$$c(z) = c_0 (1 - \lambda_1 \, z) \ldots (1 - \lambda_m z) \tag{17}$$

where

$$c_0 = \left[(-1)^m \, z_0 \, z_1 \cdots z_m\right]^{1/2}; \quad \lambda_j = \frac{1}{z_j}, \ \ j = 1, \ldots, m$$

Since $|z_j| > \sqrt{\beta}$ for $j = 1, \ldots, m$ it follows that $|\lambda_j| < 1/\sqrt{\beta}$ for $j = 1, \ldots, m$.

Using (17), we can express the factorization (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 (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 (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 (12) is

$$c(L)\,y_t = c\,(\beta L^{-1})^{-1}a_t \tag{18}$$

This can be established by using an argument paralleling that in chapter IX of [142].

To exhibit the solution in a form paralleling that of [142], we use (17) to write (18) as

$$(1-\lambda_1 L)\cdots(1-\lambda_m L)y_t = \frac{c_0^{-2}a_t}{(1-\beta\lambda_1 L^{-1})\cdots(1-\beta\lambda_m L^{-1})} \tag{19}$$

Using partial fractions, we can write the characteristic polynomial on the right side of (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 (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

$$(1-\lambda_1 L)\cdots(1-\lambda_m L)y_t = \sum_{j=1}^{m}A_j\sum_{k=0}^{\infty}(\lambda_j\beta)^k\,a_{t+k} \tag{20}$$

Equation (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, (20) is the unique solution of the Euler equation that satisfies the initial conditions and condition (12).

In effect, condition (12) compels us to solve the "unstable" roots of $h + d(\beta z^{-1})d(z)$ forward (see [142]).

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 (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 (20) converge.

We immediately see that those sums will converge under the weaker condition that $\{a_t\}$ is of exponential order less than ϕ^{-1} where $\phi = \max\{\beta\lambda_i, i = 1, \dots, m\}$.

Note that with a_t identically zero, (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 (12) is satisfied.

In fact, $\max_i |\lambda_i| < 1/\sqrt{\beta}$ is a necessary condition for (12) to hold.

Were (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 (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 (1).

Therefore it is easy to dominate a path that violates (12).

90.6 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 (16) becomes

$$\left(h + d\left(z^{-1}\right)d(z)\right) = c\left(z^{-1}\right) c\left(z\right)$$

where

$$\begin{aligned}
c\left(z\right) &= c_0(1 - \lambda_1 z) \dots (1 - \lambda_m z) \\
c_0 &= \left[(-1)^m z_0 z_1 \dots z_m\right] \\
|\lambda_j| &< 1 \text{ for } j = 1, \dots, m \\
\lambda_j &= \frac{1}{z_j} \text{ for } j = 1, \dots, m \\
z_0 &= \text{ constant}
\end{aligned}$$

The solution of the problem becomes

$$(1-\lambda_1 L)\cdots(1-\lambda_m L)y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} \lambda_j^k a_{t+k}$$

90.6.1 Transforming Discounted to Undiscounted Problem

Discounted problems can always be converted into undiscounted problems via a simple transformation.

Consider problem (1) with $0 < \beta < 1$.

Define the transformed variables

$$\tilde{a}_t = \beta^{t/2} a_t,\ \tilde{y}_t = \beta^{t/2} y_t \tag{21}$$

Then notice that $\beta^t\,[d\,(L)y_t]^2 = [\tilde{d}\,(L)\tilde{y}_t]^2$ with $\tilde{d}\,(L) = \sum_{j=0}^{m} \tilde{d}_j\, L^j$ and $\tilde{d}_j = \beta^{j/2} d_j$.

Then the original criterion function (1) is equivalent to

$$\lim_{N\to\infty} \sum_{t=0}^{N} \{\tilde{a}_t\, \tilde{y}_t - \frac{1}{2} h\, \tilde{y}_t^2 - \frac{1}{2}[\tilde{d}\,(L)\, \tilde{y}_t]^2\} \tag{22}$$

which is to be maximized over sequences $\{\tilde{y}_t,\ t\ =\ 0, ...\}$ subject to $\tilde{y}_{-1}, \cdots, \tilde{y}_{-m}$ given and $\{\tilde{a}_t,\ t = 1, ...\}$ a known bounded sequence.

The Euler equation for this problem is $[h + \tilde{d}\,(L^{-1})\,\tilde{d}\,(L)]\,\tilde{y}_t = \tilde{a}_t$.

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

$$\tilde{y}_t = \tilde{f}_1\, \tilde{y}_{t-1} + \cdots + \tilde{f}_m\, \tilde{y}_{t-m} + \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k\, \tilde{a}_{t+k}, \tag{23}$$

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 (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

$$f_j = \tilde{f}_j\, \beta^{-j/2},\ A_j = \tilde{A}_j,\ \lambda_j = \tilde{\lambda}_j\, \beta^{-1/2} \tag{24}$$

The transformations (21) and the inverse formulas (24) allow us to solve a discounted problem by first solving a related undiscounted problem.

90.7 Implementation

Here's the code that computes solutions to the LQ problem using the methods described above.

In [2]:
```
"""

Authors: Balint Skoze, Tom Sargent, John Stachurski

"""

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 ⊠ 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 ⊠_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))
            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)
        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
        """
        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)

            #--------------------------------------------
            # 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
```

90.7.1 Example

In this application, we'll have one lag, with

$$d(L)y_t = \gamma(I - L)y_t = \gamma(y_t - y_{t-1})$$

Suppose for the moment that $\gamma = 0$.

Then the intertemporal component of the LQ problem disappears, and the agent simply wants to maximize $a_t y_t - h y_t^2/2$ in each period.

This means that the agent chooses $y_t = a_t/h$.

In the following we'll set $h = 1$, so that the agent just wants to track the $\{a_t\}$ process.

However, as we increase γ, the agent gives greater weight to a smooth time path.

Hence $\{y_t\}$ evolves as a smoothed version of $\{a_t\}$.

The $\{a_t\}$ sequence we'll choose as a stationary cyclic process plus some white noise.

Here's some code that generates a plot when $\gamma = 0.8$

In [3]:
```
# 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(γ=0.8, m=1, h=1, y_m=2):

    d = γ * 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 $\gamma = {γ}$',
           xlabel='Time',
           xlim=(0, max(time))
          )
    ax.legend()
    ax.grid()
    plt.show()

plot_simulation()
```

Here's what happens when we change γ to 5.0

```
In [4]: plot_simulation(γ=5)
```

And here's $\gamma = 10$

```
In [5]: plot_simulation(γ=10)
```

90.8 Exercises

90.8.1 Exercise 1

Consider solving a discounted version ($\beta < 1$) of problem (1), as follows.

Convert (1) to the undiscounted problem (22).

Let the solution of (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

$$\tilde{y}_t = \tilde{f}_1 \tilde{y}_{t-1} + \cdots + \tilde{f}_m \tilde{y}_{t-m} + \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k} \tag{25}$$

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 (25) implies that the solution for y_t in feedback form is

$$y_t = f_1 y_{t-1} + \ldots + f_m y_{t-m} + \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} \beta^k \lambda_j^k a_{t+k}$$

where $f_j = \tilde{f}_j \beta^{-j/2}$, $A_j = \tilde{A}_j$, and $\lambda_j = \tilde{\lambda}_j \beta^{-1/2}$.

90.8.2 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" (20), giving numerical values for the coefficients.

Make sure that the boundary conditions (5) are satisfied.

(Note: this problem differs from the problem in the text in one important way: instead of $h > 0$ in (1), $h = 0$. This has an important influence on the solution.)

90.8.3 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

$$y_t = 2y_{t-1} = 2^{t+1}y_{-1} \qquad t > 0$$

90.8.4 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 (12), and so is not optimal.

Prove that the optimal solution is approximately $y_t = .5y_{t-1}$.

Chapter 91

Classical Prediction and Filtering With Linear Algebra

91.1 Contents

- Overview 91.2
- Finite Dimensional Prediction 91.3
- Combined Finite Dimensional Control and Prediction 91.4
- Infinite Horizon Prediction and Filtering Problems 91.5
- Exercises 91.6

91.2 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 pre-

diction 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 [142] [chs. IX, XIV] and Hansen and Sargent [74] 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:

```
In [1]: import numpy as np
```

91.2.1 References

Useful references include [163], [74], [123], [11], and [120].

91.3 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 (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 ε of random variables that form an orthonormal basis for the vector of random variables x.

The key insight here comes from noting that because the covariance matrix V of x is a positive definite and symmetric, there exists a (Cholesky) decomposition of V such that

$$V = L^{-1}(L^{-1})'$$

and

$$L\,V\,L' = I$$

where L and L^{-1} are both lower triangular.

Form the $T \times 1$ random vector $\varepsilon = Lx$.

The random vector ε is an orthonormal basis for x because

- L is nonsingular
- $\mathbb{E}\,\varepsilon\,\varepsilon' = L\mathbb{E}xx'L' = I$
- $x = L^{-1}\varepsilon$

It is enlightening to write out and interpret the equations $Lx = \varepsilon$ and $L^{-1}\varepsilon = x$.

First, we'll write $Lx = \varepsilon$

$$\begin{aligned} L_{11}x_1 &= \varepsilon_1 \\ L_{21}x_1 + L_{22}x_2 &= \varepsilon_2 \\ &\vdots \\ L_{T1}\,x_1 \;\ldots\; + L_{TT}x_T &= \varepsilon_T \end{aligned} \tag{1}$$

or

$$\sum_{j=0}^{t-1} L_{t,t-j}\,x_{t-j} = \varepsilon_t, \quad t = 1,\,2,\ldots T \tag{2}$$

Next, we write $L^{-1}\varepsilon = x$

$$\begin{aligned} x_1 &= L_{11}^{-1}\varepsilon_1 \\ x_2 &= L_{22}^{-1}\varepsilon_2 + L_{21}^{-1}\varepsilon_1 \\ &\vdots \\ x_T &= L_{TT}^{-1}\varepsilon_T + L_{T,T-1}^{-1}\varepsilon_{T-1} \;\ldots\; + L_{T,1}^{-1}\varepsilon_1 \end{aligned} \quad , \tag{3}$$

or

$$x_t = \sum_{j=0}^{t-1} L_{t,t-j}^{-1}\,\varepsilon_{t-j} \tag{4}$$

where $L_{i,j}^{-1}$ denotes the i,j element of L^{-1}.

From (2), it follows that ε_t is in the linear subspace spanned by $x_t,\, x_{t-1}, \ldots,\, x_1$.

From (4) it follows that that x_t is in the linear subspace spanned by $\varepsilon_t,\, \varepsilon_{t-1}, \ldots, \varepsilon_1$.

Equation (2) forms a sequence of **autoregressions** that for $t = 1, \dots, T$ express x_t as linear functions of $x_s, s = 1, \dots, t-1$ and a random variable $(L_{t,t})^{-1}\varepsilon_t$ that is orthogonal to each componenent of $x_s, s = 1, \dots, t-1$.

(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 $\varepsilon_t, \dots, \varepsilon_1$ and $x_t, \dots, x_1$ means that for $t-1 \geq m \geq 1$

$$\hat{\mathbb{E}}[x_t \mid x_{t-m}, x_{t-m-1}, \dots, x_1] = \hat{\mathbb{E}}[x_t \mid \varepsilon_{t-m}, \varepsilon_{t-m-1}, \dots, \varepsilon_1] \tag{5}$$

To proceed, it is useful to drill down and note that for $t-1 \geq m \geq 1$ we can rewrite (4) in the form of the **moving average representation**

$$x_t = \sum_{j=0}^{m-1} L_{t,t-j}^{-1} \varepsilon_{t-j} + \sum_{j=m}^{t-1} L_{t,t-j}^{-1} \varepsilon_{t-j} \tag{6}$$

Representation (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 line in that space but instead in a linear space knowns as its **orthogonal complement**.

It follows that

$$\hat{\mathbb{E}}[x_t \mid x_{t-m}, x_{t-m-1}, \dots, x_1] = \sum_{j=0}^{m-1} L_{t,t-j}^{-1} \varepsilon_{t-j}$$

91.3.1 Implementation

Here's the code that computes solutions to LQ control and filtering problems using the methods described here and in :doc: lu_tricks.

In [2]:
```
"""

Authors: Balint Skoze, Tom Sargent, John Stachurski

"""

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 ⊠ 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 ⊠_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))
        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)
        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
            """
            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)

        #--------------------------------------------
        # 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.

91.3.2 Example 1

Consider a stochastic process with moving average representation

$$x_t = (1 - 2L)\varepsilon_t$$

where ε_t is a serially uncorrelated random process with mean zero and variance unity.

If we were to use the tools associated with infinite dimensional prediction and filtering to be described below, we would use the Wiener-Kolmogorov formula (21) to compute the linear least squares forecasts $\mathbb{E}[x_{t+j} \mid x_t, x_{t-1}, ...]$, for $j = 1, 2$.

But we can do everything we want by instead using our finite dimensional tools and setting $d = r$, generating an instance of LQFilter, then invoking pertinent methods of LQFilter.

```
In [3]: 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.coefficients_of_c().

Let's check that it "flips roots" as required

```
In [4]: example.coeffs_of_c()

Out[4]: array([ 2., -1.])

In [5]: example.roots_of_characteristic()

Out[5]: (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" $Lx = \varepsilon$.

```
In [6]: 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

```
In [7]: 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

```
In [8]: 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]]
```

91.3.3 Example 2

Consider a stochastic process X_t with moving average representation

$$X_t = (1 - \sqrt{2}L^2)\varepsilon_t$$

where ε_t is a serially uncorrelated random process with mean zero and variance unity.

Let's find a Wold moving average representation for x_t that will prevail in the infinite-history context to be studied in detail below.

To do this, we'll use the Wiener-Kolomogorov formula (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

```
In [9]: 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()
```

```
Out[9]: array([ 1.41421356, -0.        , -1.        ])
```

```
In [10]: example.roots_of_characteristic()
```

```
Out[10]: (array([ 1.18920712, -1.18920712]),
          -1.4142135623731122,
          array([ 0.84089642, -0.84089642]))
```

```
In [11]: V = example.construct_V(N=8)
         print(V)
```

```
[[ 3.          0.         -1.41421356  0.          0.          0.
  0.          0.        ]
 [ 0.          3.          0.         -1.41421356  0.          0.
  0.          0.        ]
 [-1.41421356  0.          3.          0.         -1.41421356  0.
  0.          0.        ]
 [ 0.         -1.41421356  0.          3.          0.         -1.41421356
  0.          0.        ]
 [ 0.          0.         -1.41421356  0.          3.          0.
 -1.41421356  0.        ]
 [ 0.          0.          0.         -1.41421356  0.          3.
  0.         -1.41421356]
 [ 0.          0.          0.          0.         -1.41421356  0.
  3.          0.        ]
 [ 0.          0.          0.          0.          0.         -1.41421356
  0.          3.        ]]
```

```
In [12]: 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]]
```

```
In [13]: 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]]
```

91.3.4 Prediction

It immediately follows from the "orthogonality principle" of least squares (see [11] or [142] [ch. X]) that

$$\begin{aligned}\hat{\mathbb{E}}[x_t \mid x_{t-m}, x_{t-m+1}, \dots x_1] &= \sum_{j=m}^{t-1} L_{t,t-j}^{-1} \varepsilon_{t-j} \\ &= [L_{t,1}^{-1} L_{t,2}^{-1}, \dots, L_{t,t-m}^{-1} \; 0 \; 0 \dots 0] L\, x\end{aligned} \tag{7}$$

This can be interpreted as a finite-dimensional version of the Wiener-Kolmogorov m-step ahead prediction formula.

We can use (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

$$\hat{\mathbb{E}}[x \mid x_s, x_{s-1}, \dots, x_1] = L^{-1} \begin{bmatrix} I_s & 0 \\ 0 & 0_{(t-s)} \end{bmatrix} Lx \tag{8}$$

This formula will be convenient in representing the solution of control problems under uncertainty.

Equation (4) can be recognized as a finite dimensional version of a moving average representation.

Equation (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 $j > 0$, $L_{T,T-j}^{-1}$ converges to $L_{T-k,T-k-j}^{-1}$ 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.

91.4 Combined Finite Dimensional Control and Prediction

Consider the finite-dimensional control problem, maximize

$$\mathbb{E}\sum_{t=0}^{N}\left\{a_t y_t - \frac{1}{2}hy_t^2 - \frac{1}{2}[d(L)y_t]^2\right\}, \qquad h > 0$$

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 (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 (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

$$U\bar{y} = L^{-1}\tilde{U}^{-1}\begin{bmatrix} 0 & 0 \\ 0 & I_{(t+1)} \end{bmatrix}\tilde{U}\bar{a} + K\begin{bmatrix} y_{-1} \\ \vdots \\ y_{-m} \end{bmatrix}$$

91.5 Infinite Horizon Prediction and Filtering Problems

It is instructive to compare the finite-horizon formulas based on linear algebra decompositions of finite-dimensional covariance matrices with classic formulas for infinite horizon and infinite history prediction and control problems.

These classic infinite horizon formulas used the mathematics of z-transforms and lag operators.

We'll meet interesting lag operator and z-transform counterparts to our finite horizon matrix formulas.

We pose two related prediction and filtering problems.

We let Y_t be a univariate m^{th} order moving average, covariance stationary stochastic process,

$$Y_t = d(L)u_t \tag{9}$$

where $d(L) = \sum_{j=0}^{m} d_j L^j$, and u_t is a serially uncorrelated stationary random process satisfying

$$\begin{aligned} \mathbb{E}u_t &= 0 \\ \mathbb{E}u_t u_s &= \begin{cases} 1 & \text{if } t = s \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{10}$$

We impose no conditions on the zeros of $d(z)$.

A second covariance stationary process is X_t given by

$$X_t = Y_t + \varepsilon_t \tag{11}$$

where ε_t is a serially uncorrelated stationary random process with $\mathbb{E}\varepsilon_t = 0$ and $\mathbb{E}\varepsilon_t\varepsilon_s = 0$ for all distinct t and s.

We also assume that $\mathbb{E}\varepsilon_t u_s = 0$ for all t and s.

The **linear least squares prediction problem** is to find the L_2 random variable $\hat{X}_{t+j}$ among linear combinations of $\{X_t,\ X_{t-1}, ...\}$ that minimizes $\mathbb{E}(\hat{X}_{t+j} - X_{t+j})^2$.

That is, the problem is to find a $\gamma_j(L) = \sum_{k=0}^{\infty} \gamma_{jk} L^k$ such that $\sum_{k=0}^{\infty} |\gamma_{jk}|^2 < \infty$ and $\mathbb{E}[\gamma_j(L)X_t - X_{t+j}]^2$ is minimized.

The **linear least squares filtering problem** is to find a $b(L) = \sum_{j=0}^{\infty} b_j L^j$ such that $\sum_{j=0}^{\infty} |b_j|^2 < \infty$ and $\mathbb{E}[b(L)X_t - Y_t]^2$ is minimized.

Interesting versions of these problems related to the permanent income theory were studied by [120].

91.5.1 Problem Formulation

These problems are solved as follows.

The covariograms of Y and X and their cross covariogram are, respectively,

$$\begin{aligned} C_X(\tau) &= \mathbb{E}X_t X_{t-\tau} \\ C_Y(\tau) &= \mathbb{E}Y_t Y_{t-\tau} \qquad \tau = 0, \pm 1, \pm 2, ... \\ C_{Y,X}(\tau) &= \mathbb{E}Y_t X_{t-\tau} \end{aligned} \tag{12}$$

The covariance and cross-covariance generating functions are defined as

$$
\begin{aligned}
g_X(z) &= \sum_{\tau=-\infty}^{\infty} C_X(\tau) z^\tau \\
g_Y(z) &= \sum_{\tau=-\infty}^{\infty} C_Y(\tau) z^\tau \\
g_{YX}(z) &= \sum_{\tau=-\infty}^{\infty} C_{YX}(\tau) z^\tau
\end{aligned} \tag{13}
$$

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 [142] [ch. XI], it is true that

$$
\begin{aligned}
g_y(z) &= A(z)A(z^{-1}) + B(z)B(z^{-1}) \\
g_x(z) &= C(z)C(z^{-1}) + D(z)D(z^{-1}) \\
g_{yx}(z) &= A(z)C(z^{-1}) + B(z)D(z^{-1})
\end{aligned} \tag{14}
$$

Applying these formulas to (9) – (12), we have

$$
\begin{aligned}
g_Y(z) &= d(z)d(z^{-1}) \\
g_X(z) &= d(z)d(z^{-1}) + h \\
g_{YX}(z) &= d(z)d(z^{-1})
\end{aligned} \tag{15}
$$

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,

$$
X_t = c\,(L)\,\eta_t \tag{16}
$$

where $c(L) = \sum_{j=0}^{m} c_j\, L^j$, with

$$
c_0\eta_t = X_t - \hat{\mathbb{E}}[X_t | X_{t-1}, X_{t-2}, ...] \tag{17}
$$

Here $\hat{\mathbb{E}}$ is the linear least squares projection operator.

Equation (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 (17) requires that η_t lie in the closed linear space spanned by $[X_t,\ X_{t-1}, ...]$.

This will be true if and only if the zeros of $c(z)$ do not lie inside the unit circle.

It is an implication of (17) that η_t is a serially uncorrelated random process and that normalization can be imposed so that $\mathbb{E}\eta_t^2 = 1$.

Consequently, an implication of (16) is that the covariance generating function of X_t can be expressed as

$$g_X(z) = c\,(z)\,c\,(z^{-1}) \tag{18}$$

It remains to discuss how $c(L)$ is to be computed.

Combining (14) and (18) gives

$$d(z)\,d(z^{-1}) + h = c\,(z)\,c\,(z^{-1}) \tag{19}$$

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**:

$$\left[\sum_{j=-\infty}^{\infty} f_j\,L^j\right]_+ \equiv \sum_{j=0}^{\infty} f_j\,L^j \tag{20}$$

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:

$$\gamma_j\,(L) = \left[\frac{c(L)}{L^j}\right]_+ c\,(L)^{-1} \tag{21}$$

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

$$b(L) = \left[\frac{d(L)d(L^{-1})}{c(L^{-1})}\right]_+ c(L)^{-1} \tag{22}$$

Formulas (21) and (22) are discussed in detail in [164] and [142].

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 [120].

As an example of the usefulness of formula (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 (22) to obtain the answer.

Using the standard formulas (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 (22) becomes

$$b(L) = \left[\frac{c(L)}{1-\delta L^{-1}}\right]_+ c(L)^{-1} \tag{23}$$

In order to evaluate the term in the annihilation operator, we use the following result from [74].

Proposition Let

- $g(z) = \sum_{j=0}^{\infty} g_j z^j$ where $\sum_{j=0}^{\infty} |g_j|^2 < +\infty$.
- $h(z^{-1}) = (1-\delta_1 z^{-1}) ... (1-\delta_n z^{-1})$, where $|\delta_j| < 1$, for $j = 1, ..., n$.

Then

$$\left[\frac{g(z)}{h(z^{-1})}\right]_+ = \frac{g(z)}{h(z^{-1})} - \sum_{j=1}^{n} \frac{\delta_j g(\delta_j)}{\prod_{\substack{k=1 \\ k\neq j}}^{n} (\delta_j - \delta_k)} \left(\frac{1}{z-\delta_j}\right) \tag{24}$$

and, alternatively,

$$\left[\frac{g(z)}{h(z^{-1})}\right]_+ = \sum_{j=1}^{n} B_j \left(\frac{zg(z) - \delta_j g(\delta_j)}{z-\delta_j}\right) \tag{25}$$

where $B_j = 1/\prod_{\substack{k=1 \\ k+j}}^{n} (1-\delta_k/\delta_j)$.

Applying formula (25) of the proposition to evaluating (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

$$\hat{\mathbb{E}}\left[\sum_{j=0}^{\infty} \delta^j X_{t+j} | X_t,\, x_{t-1}, \ldots\right] = \left[\frac{1 - \delta c(\delta) L^{-1} c(L)^{-1}}{1 - \delta L^{-1}}\right] X_t \tag{26}$$

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

$$\mathbb{E}_0 \lim_{N \to \infty} \sum_{t-0}^{N} \beta^t \left[a_t\, y_t - \frac{1}{2}\, h y_t^2 - \frac{1}{2}\, [d(L) y_t]^2\right] \tag{27}$$

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 (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

$$(1 - \lambda_1 L) \ldots (1 - \lambda_m L) y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} (\lambda_j \beta)^k\, a_{t+k} \tag{28}$$

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 (26) and (28), we have the following solution

$$(1-\lambda_1 L)\dots(1-\lambda_m L)y_t = \sum_{j=1}^{m} A_j \left[\frac{1-\beta\lambda_j\, c(\beta\lambda_j)L^{-1}c(L)^{-1}}{1-\beta\lambda_j L^{-1}}\right] a_t$$

Blaschke factors

The following is a useful piece of mathematics underlying "root flipping".

Let $\pi(z) = \sum_{j=0}^{m} \pi_j z^j$ and let $z_1, \dots, z_k$ be the zeros of $\pi(z)$ that are inside the unit circle, $k < m$.

Then define

$$\theta(z) = \pi(z)\left(\frac{(z_1 z - 1)}{(z - z_1)}\right)\left(\frac{(z_2 z - 1)}{(z - z_2)}\right)\dots\left(\frac{(z_k z - 1)}{(z - z_k)}\right)$$

The term multiplying $\pi(z)$ is termed a "Blaschke factor".

Then it can be proved directly that

$$\theta(z^{-1})\theta(z) = \pi(z^{-1})\pi(z)$$

and that the zeros of $\theta(z)$ are not inside the unit circle.

91.6 Exercises

91.6.1 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 ε_t is a serially uncorrelated white noise with $\mathbb{E}\varepsilon_t^2 = 9$, and $\mathbb{E}\varepsilon_t u_s = 0$ for all t and s.

Find the Wold moving average representation for X_t.

Find a formula for the A_{1j}'s in

$$\mathbb{E}\widehat{X}_{t+1} \mid X_t, X_{t-1}, \dots = \sum_{j=0}^{\infty} A_{1j} X_{t-j}$$

Find a formula for the A_{2j}'s in

$$\widehat{\mathbb{E}} X_{t+2} \mid X_t, X_{t-1}, \dots = \sum_{j=0}^{\infty} A_{2j} X_{t-j}$$

91.6.2 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 ε_t be an $n \times 1$ vector white noise with mean 0 and contemporaneous covariance matrix H, where H is a positive definite matrix.

Let $X_t = Y_t + \varepsilon_t$.

Define the covariograms as $C_X(\tau) = \mathbb{E}X_t X_{t-\tau}', C_Y(\tau) = \mathbb{E}Y_t Y_{t-\tau}', C_{YX}(\tau) = \mathbb{E}Y_t X_{t-\tau}'$.

Then define the matrix covariance generating function, as in (21), only interpret all the objects in (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 [134] or [164]) of the form

$$D(z)D(z^{-1})' + H = C(z)C(z^{-1})', \quad C(z) = \sum_{j=0}^{m} C_j z^j$$

where the zeros of $|C(z)|$ do not lie inside the unit circle.

A vector Wold moving average representation of X_t is then

$$X_t = C(L)\eta_t$$

where η_t is an $(n \times 1)$ vector white noise that is "fundamental" for X_t.

That is, $X_t - \hat{\mathbb{E}}\left[X_t \mid X_{t-1}, X_{t-2} \ldots\right] = C_0\, \eta_t$.

The optimum predictor of X_{t+j} is

$$\hat{\mathbb{E}}\left[X_{t+j} \mid X_t, X_{t-1}, \ldots\right] = \left[\frac{C(L)}{L^j}\right]_+ \eta_t$$

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

$$\hat{\mathbb{E}}\left[X_{t+j} \mid X_t, X_{t-1}, \ldots\right] = \left[\frac{C(L)}{L^J}\right]_+ C(L)^{-1}\, X_t$$

Part XIII

Asset Pricing and Finance

Chapter 92

Asset Pricing I: Finite State Models

92.1 Contents

- Overview 92.2
- Pricing Models 92.3
- Prices in the Risk-Neutral Case 92.4
- Asset Prices under Risk Aversion 92.5
- Exercises 92.6
- Solutions 92.7

> "A little knowledge of geometric series goes a long way" – Robert E. Lucas, Jr.

> "Asset pricing is all about covariances" – Lars Peter Hansen

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

92.2 Overview

An asset is a claim on one or more future payoffs.

The spot price of an asset depends primarily on

- the anticipated dynamics for the stream of income accruing to the owners
- attitudes to risk
- rates of time preference

In this lecture, we consider some standard pricing models and dividend stream specifications.

We study how prices and dividend-price ratios respond in these different scenarios.

We also look at creating and pricing *derivative* assets by repackaging income streams.

Key tools for the lecture are

- formulas for predicting future values of functions of a Markov state
- a formula for predicting the discounted sum of future values of a Markov state

Let's start with some imports:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        import quantecon as qe
        from numpy.linalg import eigvals, solve
```

92.3 Pricing Models

In what follows let $\{d_t\}_{t \geq 0}$ be a stream of dividends

- A time-t **cum-dividend** asset is a claim to the stream $d_t, d_{t+1}, \ldots$.
- A time-t **ex-dividend** asset is a claim to the stream $d_{t+1}, d_{t+2}, \ldots$.

Let's look at some equations that we expect to hold for prices of assets under ex-dividend contracts (we will consider cum-dividend pricing in the exercises).

92.3.1 Risk-Neutral Pricing

Our first scenario is risk-neutral pricing.

Let $\beta = 1/(1+\rho)$ be an intertemporal discount factor, where ρ is the rate at which agents discount the future.

The basic risk-neutral asset pricing equation for pricing one unit of an ex-dividend asset is

$$p_t = \beta \mathbb{E}_t[d_{t+1} + p_{t+1}] \tag{1}$$

This is a simple "cost equals expected benefit" relationship.

Here $\mathbb{E}_t[y]$ denotes the best forecast of y, conditioned on information available at time t.

92.3.2 Pricing with Random Discount Factor

What happens if for some reason traders discount payouts differently depending on the state of the world?

Michael Harrison and David Kreps [81] and Lars Peter Hansen and Scott Richard [73] showed that in quite general settings the price of an ex-dividend asset obeys

$$p_t = \mathbb{E}_t \left[m_{t+1}(d_{t+1} + p_{t+1}) \right] \tag{2}$$

for some **stochastic discount factor** m_{t+1}.

The fixed discount factor β in (1) has been replaced by the random variable m_{t+1}.

The way anticipated future payoffs are evaluated can now depend on various random outcomes.

One example of this idea is that assets that tend to have good payoffs in bad states of the world might be regarded as more valuable.

This is because they pay well when the funds are more urgently needed.

We give examples of how the stochastic discount factor has been modeled below.

92.3.3 Asset Pricing and Covariances

Recall that, from the definition of a conditional covariance $\text{cov}_t(x_{t+1}, y_{t+1})$, we have

$$\mathbb{E}_t(x_{t+1}y_{t+1}) = \text{cov}_t(x_{t+1}, y_{t+1}) + \mathbb{E}_t x_{t+1}\mathbb{E}_t y_{t+1} \tag{3}$$

If we apply this definition to the asset pricing equation (2) we obtain

$$p_t = \mathbb{E}_t m_{t+1}\mathbb{E}_t(d_{t+1} + p_{t+1}) + \text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1}) \tag{4}$$

It is useful to regard equation (4) as a generalization of equation (1)

- In equation (1), the stochastic discount factor $m_{t+1} = \beta$, a constant.
- In equation (1), the covariance term $\text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1})$ is zero because $m_{t+1} = \beta$.

Equation (4) asserts that the covariance of the stochastic discount factor with the one period payout $d_{t+1} + p_{t+1}$ is an important determinant of the price p_t.

We give examples of some models of stochastic discount factors that have been proposed later in this lecture and also in a later lecture.

92.3.4 The Price-Dividend Ratio

Aside from prices, another quantity of interest is the **price-dividend ratio** $v_t := p_t/d_t$.

Let's write down an expression that this ratio should satisfy.

We can divide both sides of (2) by d_t to get

$$v_t = \mathbb{E}_t\left[m_{t+1}\frac{d_{t+1}}{d_t}(1 + v_{t+1})\right] \tag{5}$$

Below we'll discuss the implication of this equation.

92.4 Prices in the Risk-Neutral Case

What can we say about price dynamics on the basis of the models described above?

The answer to this question depends on

1. the process we specify for dividends
2. the stochastic discount factor and how it correlates with dividends

For now let's focus on the risk-neutral case, where the stochastic discount factor is constant, and study how prices depend on the dividend process.

92.4.1 Example 1: Constant Dividends

The simplest case is risk-neutral pricing in the face of a constant, non-random dividend stream $d_t = d > 0$.

Removing the expectation from (1) and iterating forward gives

$$\begin{aligned} p_t &= \beta(d + p_{t+1}) \\ &= \beta(d + \beta(d + p_{t+2})) \\ &\quad \vdots \\ &= \beta(d + \beta d + \beta^2 d + \cdots + \beta^{k-2} d + \beta^{k-1} p_{t+k}) \end{aligned}$$

Unless prices explode in the future, this sequence converges to

$$\bar{p} := \frac{\beta d}{1 - \beta} \tag{6}$$

This price is the equilibrium price in the constant dividend case.

Indeed, simple algebra shows that setting $p_t = \bar{p}$ for all t satisfies the equilibrium condition $p_t = \beta(d + p_{t+1})$.

92.4.2 Example 2: Dividends with Deterministic Growth Paths

Consider a growing, non-random dividend process $d_{t+1} = g d_t$ where $0 < g\beta < 1$.

While prices are not usually constant when dividends grow over time, the price dividend-ratio might be.

If we guess this, substituting $v_t = v$ into (5) as well as our other assumptions, we get $v = \beta g(1 + v)$.

Since $\beta g < 1$, we have a unique positive solution:

$$v = \frac{\beta g}{1 - \beta g}$$

The price is then

$$p_t = \frac{\beta g}{1 - \beta g} d_t$$

If, in this example, we take $g = 1 + \kappa$ and let $\rho := 1/\beta - 1$, then the price becomes

$$p_t = \frac{1 + \kappa}{\rho - \kappa} d_t$$

This is called the *Gordon formula*.

92.4.3 Example 3: Markov Growth, Risk-Neutral Pricing

Next, we consider a dividend process

$$d_{t+1} = g_{t+1} d_t \tag{7}$$

The stochastic growth factor $\{g_t\}$ is given by

$$g_t = g(X_t), \quad t = 1, 2, \ldots$$

where

1. $\{X_t\}$ is a finite Markov chain with state space S and transition probabilities

$$P(x, y) := \mathbb{P}\{X_{t+1} = y \,|\, X_t = x\} \qquad (x, y \in S)$$

1. g is a given function on S taking positive values

You can think of

- S as n possible "states of the world" and X_t as the current state.
- g as a function that maps a given state X_t into a growth factor $g_t = g(X_t)$ for the endowment.
- $\ln g_t = \ln(d_{t+1}/d_t)$ is the growth rate of dividends.

(For a refresher on notation and theory for finite Markov chains see this lecture)

The next figure shows a simulation, where

- $\{X_t\}$ evolves as a discretized AR1 process produced using Tauchen's method.
- $g_t = \exp(X_t)$, so that $\ln g_t = X_t$ is the growth rate.

In [3]:
```
mc = qe.tauchen(0.96, 0.25, n=25)
sim_length = 80

x_series = mc.simulate(sim_length, init=np.median(mc.state_values))
g_series = np.exp(x_series)
d_series = np.cumprod(g_series) # Assumes d_0 = 1

series = [x_series, g_series, d_series, np.log(d_series)]
labels = ['$X_t$', '$g_t$', '$d_t$', r'$\log \, d_t$']

fig, axes = plt.subplots(2, 2, figsize=(12, 8))
for ax, s, label in zip(axes.flatten(), series, labels):
    ax.plot(s, 'b-', lw=2, label=label)
    ax.legend(loc='upper left', frameon=False)
plt.tight_layout()
plt.show()
```

Pricing

To obtain asset prices in this setting, let's adapt our analysis from the case of deterministic growth.

In that case, we found that v is constant.

This encourages us to guess that, in the current case, v_t is constant given the state X_t.

In other words, we are looking for a fixed function v such that the price-dividend ratio satisfies $v_t = v(X_t)$.

We can substitute this guess into (5) to get

$$v(X_t) = \beta \mathbb{E}_t[g(X_{t+1})(1 + v(X_{t+1}))]$$

If we condition on $X_t = x$, this becomes

$$v(x) = \beta \sum_{y \in S} g(y)(1 + v(y))P(x, y)$$

or

$$v(x) = \beta \sum_{y \in S} K(x, y)(1 + v(y)) \quad \text{where} \quad K(x, y) := g(y)P(x, y) \tag{8}$$

Suppose that there are n possible states $x_1, \dots, x_n$.

We can then think of (8) as n stacked equations, one for each state, and write it in matrix form as

$$v = \beta K(\mathbb{1} + v) \tag{9}$$

Here

- v is understood to be the column vector $(v(x_1), \dots, v(x_n))'$.
- K is the matrix $(K(x_i, x_j))_{1 \leq i,j \leq n}$.
- $\mathbb{1}$ is a column vector of ones.

When does (9) have a unique solution?

From the Neumann series lemma and Gelfand's formula, this will be the case if βK has spectral radius strictly less than one.

In other words, we require that the eigenvalues of K be strictly less than β^{-1} in modulus.

The solution is then

$$v = (I - \beta K)^{-1} \beta K \mathbb{1} \tag{10}$$

92.4.4 Code

Let's calculate and plot the price-dividend ratio at a set of parameters.

As before, we'll generate $\{X_t\}$ as a discretized AR1 process and set $g_t = \exp(X_t)$.

Here's the code, including a test of the spectral radius condition

In [4]:
```
n = 25  # Size of state space
β = 0.9
mc = qe.tauchen(0.96, 0.02, n=n)

K = mc.P * np.exp(mc.state_values)

warning_message = "Spectral radius condition fails"
assert np.max(np.abs(eigvals(K))) < 1 / β,  warning_message

I = np.identity(n)
v = solve(I - β * K, β * K @ np.ones(n))

fig, ax = plt.subplots(figsize=(12, 8))
ax.plot(mc.state_values, v, 'g-o', lw=2, alpha=0.7, label='$v$')
ax.set_ylabel("price-dividend ratio")
ax.set_xlabel("state")
ax.legend(loc='upper left')
plt.show()
```

Why does the price-dividend ratio increase with the state?

The reason is that this Markov process is positively correlated, so high current states suggest high future states.

Moreover, dividend growth is increasing in the state.

The anticipation of high future dividend growth leads to a high price-dividend ratio.

92.5 Asset Prices under Risk Aversion

Now let's turn to the case where agents are risk averse.

We'll price several distinct assets, including

- The price of an endowment stream
- A consol (a type of bond issued by the UK government in the 19th century)
- Call options on a consol

92.5.1 Pricing a Lucas Tree

Let's start with a version of the celebrated asset pricing model of Robert E. Lucas, Jr. [109].

As in [109], suppose that the stochastic discount factor takes the form

$$m_{t+1} = \beta \frac{u'(c_{t+1})}{u'(c_t)} \tag{11}$$

where u is a concave utility function and c_t is time t consumption of a representative consumer.

(A derivation of this expression is given in a later lecture)

Assume the existence of an endowment that follows (7).

The asset being priced is a claim on the endowment process.

Following [109], suppose further that in equilibrium, consumption is equal to the endowment, so that $d_t = c_t$ for all t.

For utility, we'll assume the **constant relative risk aversion** (CRRA) specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \text{ with } \gamma > 0 \tag{12}$$

When $\gamma = 1$ we let $u(c) = \ln c$.

Inserting the CRRA specification into (11) and using $c_t = d_t$ gives

$$m_{t+1} = \beta \left(\frac{c_{t+1}}{c_t} \right)^{-\gamma} = \beta g_{t+1}^{-\gamma} \tag{13}$$

Substituting this into (5) gives the price-dividend ratio formula

$$v(X_t) = \beta \mathbb{E}_t \left[g(X_{t+1})^{1-\gamma} (1 + v(X_{t+1})) \right]$$

Conditioning on $X_t = x$, we can write this as

$$v(x) = \beta \sum_{y \in S} g(y)^{1-\gamma} (1 + v(y)) P(x, y)$$

If we let

$$J(x, y) := g(y)^{1-\gamma} P(x, y)$$

then we can rewrite in vector form as

$$v = \beta J (\mathbb{1} + v)$$

Assuming that the spectral radius of J is strictly less than β^{-1}, this equation has the unique solution

$$v = (I - \beta J)^{-1} \beta J \mathbb{1} \tag{14}$$

We will define a function tree_price to solve for v given parameters stored in the class AssetPriceModel

```
In [5]: class AssetPriceModel:
            """
            A class that stores the primitives of the asset pricing model.

            Parameters
            ----------
```

```
    β : scalar, float
        Discount factor
    mc : MarkovChain
       Contains the transition matrix and set of state values for the state
        process
    γ : scalar(float)
        Coefficient of risk aversion
    g : callable
        The function mapping states to growth rates

    """
    def __init__(self, β=0.96, mc=None, γ=2.0, g=np.exp):
        self.β, self.γ = β, γ
        self.g = g

        # A default process for the Markov chain
        if mc is None:
            self.ρ = 0.9
            self.σ = 0.02
            self.mc = qe.tauchen(self.ρ, self.σ, n=25)
        else:
            self.mc = mc

        self.n = self.mc.P.shape[0]

    def test_stability(self, Q):
        """
        Stability test for a given matrix Q.
        """
        sr = np.max(np.abs(eigvals(Q)))
        if not sr < 1 / self.β:
            msg = f"Spectral radius condition failed with radius = {sr}"
            raise ValueError(msg)


def tree_price(ap):
    """
    Computes the price-dividend ratio of the Lucas tree.

    Parameters
    ----------
    ap: AssetPriceModel
        An instance of AssetPriceModel containing primitives

    Returns
    -------
    v : array_like(float)
        Lucas tree price-dividend ratio

    """
    # Simplify names, set up matrices
    β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
    J = P * ap.g(y)**(1 - γ)

    # Make sure that a unique solution exists
    ap.test_stability(J)

    # Compute v
```

```
    I = np.identity(ap.n)
    Ones = np.ones(ap.n)
    v = solve(I - β * J, β * J @ Ones)

    return v
```

Here's a plot of v as a function of the state for several values of γ, with a positively correlated Markov process and $g(x) = \exp(x)$

```
In [6]: γs = [1.2, 1.4, 1.6, 1.8, 2.0]
        ap = AssetPriceModel()
        states = ap.mc.state_values

        fig, ax = plt.subplots(figsize=(12, 8))

        for γ in γs:
            ap.γ = γ
            v = tree_price(ap)
            ax.plot(states, v, lw=2, alpha=0.6, label=rf"$\gamma = {γ}$")

        ax.set_title('Price-divdend ratio as a function of the state')
        ax.set_ylabel("price-dividend ratio")
        ax.set_xlabel("state")
        ax.legend(loc='upper right')
        plt.show()
```

Notice that v is decreasing in each case.

This is because, with a positively correlated state process, higher states suggest higher future consumption growth.

In the stochastic discount factor (13), higher growth decreases the discount factor, lowering the weight placed on future returns.

Special Cases

In the special case $\gamma = 1$, we have $J = P$.

Recalling that $P^i \mathbb{1} = \mathbb{1}$ for all i and applying Neumann's geometric series lemma, we are led to

$$v = \beta(I - \beta P)^{-1}\mathbb{1} = \beta \sum_{i=0}^{\infty} \beta^i P^i \mathbb{1} = \beta \frac{1}{1-\beta}\mathbb{1}$$

Thus, with log preferences, the price-dividend ratio for a Lucas tree is constant.

Alternatively, if $\gamma = 0$, then $J = K$ and we recover the risk-neutral solution (10).

This is as expected, since $\gamma = 0$ implies $u(c) = c$ (and hence agents are risk-neutral).

92.5.2 A Risk-Free Consol

Consider the same pure exchange representative agent economy.

A risk-free consol promises to pay a constant amount $\zeta > 0$ each period.

Recycling notation, let p_t now be the price of an ex-coupon claim to the consol.

An ex-coupon claim to the consol entitles the owner at the end of period t to

- ζ in period $t+1$, plus
- the right to sell the claim for p_{t+1} next period

The price satisfies (2) with $d_t = \zeta$, or

$$p_t = \mathbb{E}_t \left[m_{t+1}(\zeta + p_{t+1}) \right]$$

We maintain the stochastic discount factor (13), so this becomes

$$p_t = \mathbb{E}_t \left[\beta g_{t+1}^{-\gamma}(\zeta + p_{t+1}) \right] \tag{15}$$

Guessing a solution of the form $p_t = p(X_t)$ and conditioning on $X_t = x$, we get

$$p(x) = \beta \sum_{y \in S} g(y)^{-\gamma}(\zeta + p(y))P(x, y)$$

Letting $M(x, y) = P(x, y)g(y)^{-\gamma}$ and rewriting in vector notation yields the solution

$$p = (I - \beta M)^{-1}\beta M \zeta \mathbb{1} \tag{16}$$

The above is implemented in the function consol_price.

```
In [7]: def consol_price(ap, ζ):
        """
```

```
    Computes price of a consol bond with payoff ζ

    Parameters
    ----------
    ap: AssetPriceModel
        An instance of AssetPriceModel containing primitives

    ζ : scalar(float)
        Coupon of the console

    Returns
    -------
    p : array_like(float)
        Console bond prices

    """
    # Simplify names, set up matrices
    β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
    M = P * ap.g(y)**(- γ)

    # Make sure that a unique solution exists
    ap.test_stability(M)

    # Compute price
    I = np.identity(ap.n)
    Ones = np.ones(ap.n)
    p = solve(I - β * M, β * ζ * M @ Ones)

    return p
```

92.5.3 Pricing an Option to Purchase the Consol

Let's now price options of varying maturity that give the right to purchase a consol at a price p_S.

An Infinite Horizon Call Option

We want to price an infinite horizon option to purchase a consol at a price p_S.

The option entitles the owner at the beginning of a period either to

1. purchase the bond at price p_S now, or
2. Not to exercise the option now but to retain the right to exercise it later

Thus, the owner either *exercises* the option now or chooses *not to exercise* and wait until next period.

This is termed an infinite-horizon *call option* with *strike price* p_S.

The owner of the option is entitled to purchase the consol at the price p_S at the beginning of any period, after the coupon has been paid to the previous owner of the bond.

The fundamentals of the economy are identical with the one above, including the stochastic discount factor and the process for consumption.

Let $w(X_t, p_S)$ be the value of the option when the time t growth state is known to be X_t but *before* the owner has decided whether or not to exercise the option at time t (i.e., today).

Recalling that $p(X_t)$ is the value of the consol when the initial growth state is X_t, the value of the option satisfies

$$w(X_t, p_S) = \max\left\{\beta\,\mathbb{E}_t \frac{u'(c_{t+1})}{u'(c_t)} w(X_{t+1}, p_S),\ p(X_t) - p_S\right\}$$

The first term on the right is the value of waiting, while the second is the value of exercising now.

We can also write this as

$$w(x, p_S) = \max\left\{\beta \sum_{y \in S} P(x,y) g(y)^{-\gamma} w(y, p_S),\ p(x) - p_S\right\} \tag{17}$$

With $M(x,y) = P(x,y)g(y)^{-\gamma}$ and w as the vector of values $(w(x_i), p_S)_{i=1}^n$, we can express (17) as the nonlinear vector equation

$$w = \max\{\beta M w,\ p - p_S \mathbb{1}\} \tag{18}$$

To solve (18), form the operator T mapping vector w into vector Tw via

$$Tw = \max\{\beta M w,\ p - p_S \mathbb{1}\}$$

Start at some initial w and iterate to convergence with T.

We can find the solution with the following function call_option

```
In [8]: def call_option(ap, ζ, p_s, ϵ=1e-7):
            """
            Computes price of a call option on a consol bond.

            Parameters
            ----------
            ap: AssetPriceModel
                An instance of AssetPriceModel containing primitives

            ζ : scalar(float)
                Coupon of the console

            p_s : scalar(float)
                Strike price

            ϵ : scalar(float), optional(default=1e-8)
                Tolerance for infinite horizon problem

            Returns
            -------
            w : array_like(float)
                Infinite horizon call option prices

            """
```

```
    # Simplify names, set up matrices
    β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
    M = P * ap.g(y)**(- γ)

    # Make sure that a unique consol price exists
    ap.test_stability(M)

    # Compute option price
    p = consol_price(ap, ζ)
    w = np.zeros(ap.n)
    error = ϵ + 1
    while error > ϵ:
        # Maximize across columns
        w_new = np.maximum(β * M @ w, p - p_s)
        # Find maximal difference of each component and update
        error = np.amax(np.abs(w - w_new))
        w = w_new

    return w
```

Here's a plot of w compared to the consol price when $P_S = 40$

```
In [9]: ap = AssetPriceModel(β=0.9)
        ζ = 1.0
        strike_price = 40

        x = ap.mc.state_values
        p = consol_price(ap, ζ)
        w = call_option(ap, ζ, strike_price)

        fig, ax = plt.subplots(figsize=(12, 8))
        ax.plot(x, p, 'b-', lw=2, label='consol price')
        ax.plot(x, w, 'g-', lw=2, label='value of call option')
        ax.set_xlabel("state")
        ax.legend(loc='upper right')
        plt.show()
```

In large states, the value of the option is close to zero.

This is despite the fact the Markov chain is irreducible and low states — where the consol prices are high — will eventually be visited.

The reason is that $\beta = 0.9$, so the future is discounted relatively rapidly.

92.5.4 Risk-Free Rates

Let's look at risk-free interest rates over different periods.

The One-period Risk-free Interest Rate

As before, the stochastic discount factor is $m_{t+1} = \beta g_{t+1}^{-\gamma}$.

It follows that the reciprocal R_t^{-1} of the gross risk-free interest rate R_t in state x is

$$\mathbb{E}_t m_{t+1} = \beta \sum_{y \in S} P(x, y) g(y)^{-\gamma}$$

We can write this as

$$m_1 = \beta M \mathbb{1}$$

where the i-th element of m_1 is the reciprocal of the one-period gross risk-free interest rate in state x_i.

Other Terms

Let m_j be an $n \times 1$ vector whose i th component is the reciprocal of the j -period gross risk-free interest rate in state x_i.

Then $m_1 = \beta M$, and $m_{j+1} = M m_j$ for $j \geq 1$.

92.6 Exercises

92.6.1 Exercise 1

In the lecture, we considered **ex-dividend assets**.

A **cum-dividend** asset is a claim to the stream $d_t, d_{t+1}, \ldots$.

Following (1), find the risk-neutral asset pricing equation for one unit of a cum-dividend asset.

With a constant, non-random dividend stream $d_t = d > 0$, what is the equilibrium price of a cum-dividend asset?

With a growing, non-random dividend process $d_t = g d_t$ where $0 < g\beta < 1$, what is the equilibrium price of a cum-dividend asset?

92.6.2 Exercise 2

Consider the following primitives

```
In [10]: n = 5
         P = 0.0125 * np.ones((n, n))
         P += np.diag(0.95 - 0.0125 * np.ones(5))
         # State values of the Markov chain
         s = np.array([0.95, 0.975, 1.0, 1.025, 1.05])
         γ = 2.0
         β = 0.94
```

Let g be defined by $g(x) = x$ (that is, g is the identity map).

Compute the price of the Lucas tree.

Do the same for

- the price of the risk-free consol when $\zeta = 1$
- the call option on the consol when $\zeta = 1$ and $p_S = 150.0$

92.6.3 Exercise 3

Let's consider finite horizon call options, which are more common than the infinite horizon variety.

Finite horizon options obey functional equations closely related to (17).

A k period option expires after k periods.

If we view today as date zero, a k period option gives the owner the right to exercise the option to purchase the risk-free consol at the strike price p_S at dates $0, 1, \ldots, k-1$.

The option expires at time k.

Thus, for $k = 1, 2, \ldots$, let $w(x, k)$ be the value of a k-period option.

It obeys

$$w(x, k) = \max \left\{ \beta \sum_{y \in S} P(x, y) g(y)^{-\gamma} w(y, k-1),\ p(x) - p_S \right\}$$

where $w(x, 0) = 0$ for all x.

We can express the preceding as the sequence of nonlinear vector equations

$$w_k = \max\{\beta M w_{k-1},\ p - p_S \mathbb{1}\} \quad k = 1, 2, \ldots \quad \text{with } w_0 = 0$$

Write a function that computes w_k for any given k.

Compute the value of the option with `k = 5` and `k = 25` using parameter values as in Exercise 1.

Is one higher than the other? Can you give intuition?

92.7 Solutions

92.7.1 Exercise 1

For a cum-dividend asset, the basic risk-neutral asset pricing equation is

$$p_t = d_t + \beta \mathbb{E}_t[p_{t+1}]$$

With constant dividends, the equilibrium price is

$$p_t = \frac{1}{1-\beta} d_t$$

With a growing, non-random dividend process, the equilibrium price is

$$p_t = \frac{1}{1-\beta g} d_t$$

92.7.2 Exercise 2

First, let's enter the parameters:

```
In [11]: n = 5
         P = 0.0125 * np.ones((n, n))
         P += np.diag(0.95 - 0.0125 * np.ones(5))
         s = np.array([0.95, 0.975, 1.0, 1.025, 1.05])  # State values
```

```
mc = qe.MarkovChain(P, state_values=s)

γ = 2.0
β = 0.94
ζ = 1.0
p_s = 150.0
```

Next, we'll create an instance of `AssetPriceModel` to feed into the functions

```
In [12]: apm = AssetPriceModel(β=β, mc=mc, γ=γ, g=lambda x: x)
```

Now we just need to call the relevant functions on the data:

```
In [13]: tree_price(apm)

Out[13]: array([29.47401578, 21.93570661, 17.57142236, 14.72515002, 12.72221763])

In [14]: consol_price(apm, ζ)

Out[14]: array([753.87100476, 242.55144082, 148.67554548, 109.25108965,
                 87.56860139])

In [15]: call_option(apm, ζ, p_s)

Out[15]: array([603.87100476, 176.8393343 , 108.67734499,  80.05179254,
                 64.30843748])
```

Let's show the last two functions as a plot

```
In [16]: fig, ax = plt.subplots()
         ax.plot(s, consol_price(apm, ζ), label='consol')
         ax.plot(s, call_option(apm, ζ, p_s), label='call option')
         ax.legend()
         plt.show()
```

92.7.3 Exercise 3

Here's a suitable function:

```
In [17]: def finite_horizon_call_option(ap, ζ, p_s, k):
             """
             Computes k period option value.
             """
             # Simplify names, set up matrices
             β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
             M = P * ap.g(y)**(- γ)

             # Make sure that a unique solution exists
             ap.test_stability(M)


             # Compute option price
             p = consol_price(ap, ζ)
             w = np.zeros(ap.n)
             for i in range(k):
                 # Maximize across columns
                 w = np.maximum(β * M @ w, p - p_s)

             return w
```

Now let's compute the option values at `k=5` and `k=25`

```
In [18]: fig, ax = plt.subplots()
         for k in [5, 25]:
             w = finite_horizon_call_option(apm, ζ, p_s, k)
             ax.plot(s, w, label=rf'$k = {k}$')
         ax.legend()
         plt.show()
```

Not surprisingly, the option has greater value with larger k.

This is because the owner has a longer time horizon over which he or she may exercise the option.

Chapter 93

Asset Pricing II: The Lucas Asset Pricing Model

93.1 Contents

- Overview 93.2
- The Lucas Model 93.3
- Exercises 93.4
- Solutions 93.5

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install interpolation
```

93.2 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 [109] 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:

```
In [2]: 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
```

93.3 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.

93.3.1 Basic Setup

Let's review the setup.

Assets

There is a single "productive unit" that costlessly generates a sequence of consumption goods $\{y_t\}_{t=0}^{\infty}$.

Another way to view $\{y_t\}_{t=0}^{\infty}$ is as a *consumption endowment* for this economy.

We will assume that this endowment is Markovian, following the exogenous process

$$y_{t+1} = G(y_t, \xi_{t+1})$$

Here $\{\xi_t\}$ is an IID shock sequence with known distribution ϕ and $y_t \geq 0$.

An asset is a claim on all or part of this endowment stream.

The consumption goods $\{y_t\}_{t=0}^{\infty}$ are nonstorable, so holding assets is the only way to transfer wealth into the future.

For the purposes of intuition, it's common to think of the productive unit as a "tree" that produces fruit.

Based on this idea, a "Lucas tree" is a claim on the consumption endowment.

Consumers

A representative consumer ranks consumption streams $\{c_t\}$ according to the time separable utility functional

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{1}$$

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.

93.3.2 Pricing a Lucas Tree

What is an appropriate price for a claim on the consumption endowment?

We'll price an *ex-dividend* claim, meaning that

- the seller retains this period's dividend
- the buyer pays p_t today to purchase a claim on
  - y_{t+1} and
  - the right to sell the claim tomorrow at price p_{t+1}

Since this is a competitive model, the first step is to pin down consumer behavior, taking prices as given.

Next, we'll impose equilibrium constraints and try to back out prices.

In the consumer problem, the consumer's control variable is the share π_t of the claim held in each period.

Thus, the consumer problem is to maximize (1) subject to

$$c_t + \pi_{t+1} p_t \leq \pi_t y_t + \pi_t p_t$$

along with $c_t \geq 0$ and $0 \leq \pi_t \leq 1$ at each t.

The decision to hold share π_t is actually made at time $t - 1$.

But this value is inherited as a state variable at time t, which explains the choice of subscript.

The Dynamic Program

We can write the consumer problem as a dynamic programming problem.

Our first observation is that prices depend on current information, and current information is really just the endowment process up until the current period.

In fact, the endowment process is Markovian, so that the only relevant information is the current state $y \in \mathbb{R}_+$ (dropping the time subscript).

This leads us to guess an equilibrium where price is a function p of y.

Remarks on the solution method

- Since this is a competitive (read: price taking) model, the consumer will take this function p as given.
- In this way, we determine consumer behavior given p and then use equilibrium conditions to recover p.
- This is the standard way to solve competitive equilibrium models.

Using the assumption that price is a given function p of y, we write the value function and constraint as

$$v(\pi, y) = \max_{c,\pi'} \left\{ u(c) + \beta \int v(\pi', G(y,z))\phi(dz) \right\}$$

subject to

$$c + \pi' p(y) \leq \pi y + \pi p(y) \tag{2}$$

We can invoke the fact that utility is increasing to claim equality in (2) and hence eliminate the constraint, obtaining

$$v(\pi, y) = \max_{\pi'} \left\{ u[\pi(y + p(y)) - \pi' p(y)] + \beta \int v(\pi', G(y,z))\phi(dz) \right\} \tag{3}$$

The solution to this dynamic programming problem is an optimal policy expressing either π' or c as a function of the state (π, y).

- Each one determines the other, since $c(\pi, y) = \pi(y + p(y)) - \pi'(\pi, y)p(y)$

Next Steps

What we need to do now is determine equilibrium prices.

It seems that to obtain these, we will have to

1. Solve this two-dimensional dynamic programming problem for the optimal policy.
2. Impose equilibrium constraints.
3. Solve out for the price function $p(y)$ directly.

However, as Lucas showed, there is a related but more straightforward way to do this.

Equilibrium Constraints

Since the consumption good is not storable, in equilibrium we must have $c_t = y_t$ for all t.

In addition, since there is one representative consumer (alternatively, since all consumers are identical), there should be no trade in equilibrium.

In particular, the representative consumer owns the whole tree in every period, so $\pi_t = 1$ for all t.

Prices must adjust to satisfy these two constraints.

The Equilibrium Price Function

Now observe that the first-order condition for (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 (3) with respect to π, yielding

$$v_1'(\pi, y) = u'(c)(y + p(y))$$

Next, we impose the equilibrium constraints while combining the last two equations to get

$$p(y) = \beta \int \frac{u'[G(y,z)]}{u'(y)} [G(y,z) + p(G(y,z))] \phi(dz) \tag{4}$$

In sequential rather than functional notation, we can also write this as

$$p_t = \mathbb{E}_t \left[\beta \frac{u'(c_{t+1})}{u'(c_t)} (y_{t+1} + p_{t+1}) \right] \tag{5}$$

This is the famous consumption-based asset pricing equation.

Before discussing it further we want to solve out for prices.

93.3.3 Solving the Model

Equation (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

$$f(y) := u'(y) p(y) \tag{6}$$

so that (4) becomes

$$f(y) = h(y) + \beta \int f[G(y,z)] \phi(dz) \tag{7}$$

Here $h(y) := \beta \int u'[G(y,z)] G(y,z) \phi(dz)$ is a function that depends only on the primitives.

Equation (7) is a functional equation in f.

The plan is to solve out for f and convert back to p via (6).

To solve (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

$$(Tf)(y) = h(y) + \beta \int f[G(y,z)] \phi(dz) \tag{8}$$

In what follows, we refer to T as the Lucas operator.

The reason we do this is that a solution to (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

$$\|Tf - Tg\| \leq \alpha \|f - g\|, \qquad \forall\, f, g \in cb\mathbb{R}_+ \tag{9}$$

Here $\|h\| := \sup_{x \in \mathbb{R}_+} |h(x)|$.

To see that (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 (9) with $\alpha := \beta$.

93.3.4 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

In [3]:
```
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

In [4]:
```
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.

```
In [5]: def solve_model(tree, tol=1e-6, max_iter=500):
            """
            Compute the equilibrium price function associated with Lucas
            tree

            * tree is an instance of LucasTree

            """
            # Simplify notation
            grid, grid_size = tree.grid, tree.grid_size
            γ = tree.γ

            T = operator_factory(tree)

            i = 0
            f = np.ones_like(grid)  # Initial guess of f
            error = tol + 1
            while error > tol and i < max_iter:
                Tf = T(f)
                error = np.max(np.abs(Tf - f))
                f = Tf
                i += 1

            price = f * grid**γ  # Back out price vector

            return price
```

Solving the model and plotting the resulting price function

```
In [6]: 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.

93.4 Exercises

93.4.1 Exercise 1

Replicate the figure to show how discount factors affect prices.

93.5 Solutions

93.5.1 Exercise 1

In [7]:
```
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 94

Asset Pricing III: Incomplete Markets

94.1 Contents

- Overview 94.2
- Structure of the Model 94.3
- Solving the Model 94.4
- Exercises 94.5
- Solutions 94.6

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

94.2 Overview

This lecture describes a version of a model of Harrison and Kreps [80].

The model determines the price of a dividend-yielding asset that is traded by two types of self-interested investors.

The model features

- heterogeneous beliefs
- incomplete markets
- short sales constraints, and possibly ...
- (leverage) limits on an investor's ability to borrow in order to finance purchases of a risky asset

Let's start with some standard imports:

```
In [2]: import numpy as np
        import quantecon as qe
        import scipy.linalg as la
```

94.2.1 References

Prior to reading the following, you might like to review our lectures on

- Markov chains
- Asset pricing with finite state space

94.2.2 Bubbles

Economists differ in how they define a *bubble*.

The Harrison-Kreps model illustrates the following notion of a bubble that attracts many economists:

> *A component of an asset price can be interpreted as a bubble when all investors agree that the current price of the asset exceeds what they believe the asset's underlying dividend stream justifies.*

94.3 Structure of the Model

The model simplifies by ignoring alterations in the distribution of wealth among investors having different beliefs about the fundamentals that determine asset payouts.

There is a fixed number A of shares of an asset.

Each share entitles its owner to a stream of dividends $\{d_t\}$ governed by a Markov chain defined on a state space $S \in \{0, 1\}$.

The dividend obeys

$$d_t = \begin{cases} 0 & \text{if } s_t = 0 \\ 1 & \text{if } s_t = 1 \end{cases}$$

The owner of a share at the beginning of time t is entitled to the dividend paid at time t.

The owner of the share at the beginning of time t is also entitled to sell the share to another investor during time t.

Two types $h = a, b$ of investors differ only in their beliefs about a Markov transition matrix P with typical element

$$P(i, j) = \mathbb{P}\{s_{t+1} = j \mid s_t = i\}$$

Investors of type a believe the transition matrix

$$P_a = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

Investors of type b think the transition matrix is

$$P_b = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

The stationary (i.e., invariant) distributions of these two matrices can be calculated as follows:

```
In [3]: qa = np.array([[1/2, 1/2], [2/3, 1/3]])
        qb = np.array([[2/3, 1/3], [1/4, 3/4]])
        mca = qe.MarkovChain(qa)
        mcb = qe.MarkovChain(qb)
        mca.stationary_distributions
```

```
Out[3]: array([[0.57142857, 0.42857143]])
```

```
In [4]: mcb.stationary_distributions
```

```
Out[4]: array([[0.42857143, 0.57142857]])
```

The stationary distribution of P_a is approximately $\pi_A = \begin{bmatrix} .57 & .43 \end{bmatrix}$.

The stationary distribution of P_b is approximately $\pi_B = \begin{bmatrix} .43 & .57 \end{bmatrix}$.

94.3.1 Ownership Rights

An owner of the asset at the end of time t is entitled to the dividend at time $t+1$ and also has the right to sell the asset at time $t+1$.

Both types of investors are risk-neutral and both have the same fixed discount factor $\beta \in (0,1)$.

In our numerical example, we'll set $\beta = .75$, just as Harrison and Kreps did.

We'll eventually study the consequences of two different assumptions about the number of shares A relative to the resources that our two types of investors can invest in the stock.

1. Both types of investors have enough resources (either wealth or the capacity to borrow) so that they can purchase the entire available stock of the asset Section **??**.

2. No single type of investor has sufficient resources to purchase the entire stock.

Case 1 is the case studied in Harrison and Kreps.

In case 2, both types of investors always hold at least some of the asset.

94.3.2 Short Sales Prohibited

No short sales are allowed.

This matters because it limits pessimists from expressing their opinions.

- They can express their views by selling their shares.
- They cannot express their pessimism more loudly by artificially "manufacturing shares" – that is, they cannot borrow shares from more optimistic investors and sell them immediately.

94.3.3 Optimism and Pessimism

The above specifications of the perceived transition matrices P_a and P_b, taken directly from Harrison and Kreps, build in stochastically alternating temporary optimism and pessimism.

Remember that state 1 is the high dividend state.

- In state 0, a type a agent is more optimistic about next period's dividend than a type b agent.
- In state 1, a type b agent is more optimistic about next period's dividend.

However, the stationary distributions $\pi_A = \begin{bmatrix} .57 & .43 \end{bmatrix}$ and $\pi_B = \begin{bmatrix} .43 & .57 \end{bmatrix}$ tell us that a type B person is more optimistic about the dividend process in the long run than is a type A person.

Transition matrices for the temporarily optimistic and pessimistic investors are constructed as follows.

Temporarily optimistic investors (i.e., the investor with the most optimistic beliefs in each state) believe the transition matrix

$$P_o = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

Temporarily pessimistic believe the transition matrix

$$P_p = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

We'll return to these matrices and their significance in the exercise.

94.3.4 Information

Investors know a price function mapping the state s_t at t into the equilibrium price $p(s_t)$ that prevails in that state.

This price function is endogenous and to be determined below.

When investors choose whether to purchase or sell the asset at t, they also know s_t.

94.4 Solving the Model

Now let's turn to solving the model.

This amounts to determining equilibrium prices under the different possible specifications of beliefs and constraints listed above.

In particular, we compare equilibrium price functions under the following alternative assumptions about beliefs:

1. There is only one type of agent, either a or b.
2. There are two types of agents differentiated only by their beliefs. Each type of agent has sufficient resources to purchase all of the asset (Harrison and Kreps's setting).

3. There are two types of agents with different beliefs, but because of limited wealth and/or limited leverage, both types of investors hold the asset each period.

s_t	0	1
p_a	1.33	1.22
p_b	1.45	1.91
p_o	1.85	2.08
p_p	1	1
$\hat{p}_a$	1.85	1.69
$\hat{p}_b$	1.69	2.08

Here

- p_a is the equilibrium price function under homogeneous beliefs P_a
- p_b is the equilibrium price function under homogeneous beliefs P_b
- p_o is the equilibrium price function under heterogeneous beliefs with optimistic marginal investors
- p_p is the equilibrium price function under heterogeneous beliefs with pessimistic marginal investors
- $\hat{p}_a$ is the amount type a investors are willing to pay for the asset
- $\hat{p}_b$ is the amount type b investors are willing to pay for the asset

We'll explain these values and how they are calculated one row at a time.

94.4.1 Single Belief Prices

We'll start by pricing the asset under homogeneous beliefs.

(This is the case treated in the lecture on asset pricing with finite Markov states)

Suppose that there is only one type of investor, either of type a or b, and that this investor always "prices the asset".

Let $p_h = \begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix}$ be the equilibrium price vector when all investors are of type h.

The price today equals the expected discounted value of tomorrow's dividend and tomorrow's price of the asset:

$$p_h(s) = \beta \left(P_h(s,0)(0 + p_h(0)) + P_h(s,1)(1 + p_h(1)) \right), \quad s = 0, 1$$

These equations imply that the equilibrium price vector is

$$\begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix} = \beta [I - \beta P_h]^{-1} P_h \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{1}$$

The first two rows of the table report $p_a(s)$ and $p_b(s)$.

Here's a function that can be used to compute these values

```
In [5]: """
```

```
Authors: Chase Coleman, Tom Sargent

"""

def price_single_beliefs(transition, dividend_payoff, β=.75):
    """
    Function to Solve Single Beliefs
    """
    # First compute inverse piece
    imbq_inv = la.inv(np.eye(transition.shape[0]) - β * transition)

    # Next compute prices
    prices = β * imbq_inv @ transition @ dividend_payoff

    return prices
```

Single Belief Prices as Benchmarks

These equilibrium prices under homogeneous beliefs are important benchmarks for the subsequent analysis.

- $p_h(s)$ tells what investor h thinks is the "fundamental value" of the asset.
- Here "fundamental value" means the expected discounted present value of future dividends.

We will compare these fundamental values of the asset with equilibrium values when traders have different beliefs.

94.4.2 Pricing under Heterogeneous Beliefs

There are several cases to consider.

The first is when both types of agents have sufficient wealth to purchase all of the asset themselves.

In this case, the marginal investor who prices the asset is the more optimistic type so that the equilibrium price $\bar{p}$ satisfies Harrison and Kreps's key equation:

$$\bar{p}(s) = \beta \max \left\{ P_a(s,0)\bar{p}(0) + P_a(s,1)(1+\bar{p}(1)),\ P_b(s,0)\bar{p}(0) + P_b(s,1)(1+\bar{p}(1)) \right\} \tag{2}$$

for $s = 0, 1$.

The marginal investor who prices the asset in state s is of type a if

$$P_a(s,0)\bar{p}(0) + P_a(s,1)(1+\bar{p}(1)) > P_b(s,0)\bar{p}(0) + P_b(s,1)(1+\bar{p}(1))$$

The marginal investor is of type b if

$$P_a(s,1)\bar{p}(0) + P_a(s,1)(1+\bar{p}(1)) < P_b(s,1)\bar{p}(0) + P_b(s,1)(1+\bar{p}(1))$$

Thus the marginal investor is the (temporarily) optimistic type.

Equation (2) is a functional equation that, like a Bellman equation, can be solved by

- starting with a guess for the price vector $\bar{p}$ and
- iterating to convergence on the operator that maps a guess $\bar{p}^j$ into an updated guess $\bar{p}^{j+1}$ defined by the right side of (2), namely

$$\bar{p}^{j+1}(s) = \beta \max \left\{ P_a(s,0)\bar{p}^j(0) + P_a(s,1)(1+\bar{p}^j(1)),\ P_b(s,0)\bar{p}^j(0) + P_b(s,1)(1+\bar{p}^j(1)) \right\} \quad (3)$$

for $s = 0, 1$.

The third row of the table reports equilibrium prices that solve the functional equation when $\beta = .75$.

Here the type that is optimistic about s_{t+1} prices the asset in state s_t.

It is instructive to compare these prices with the equilibrium prices for the homogeneous belief economies that solve under beliefs P_a and P_b.

Equilibrium prices $\bar{p}$ in the heterogeneous beliefs economy exceed what any prospective investor regards as the fundamental value of the asset in each possible state.

Nevertheless, the economy recurrently visits a state that makes each investor want to purchase the asset for more than he believes its future dividends are worth.

The reason is that he expects to have the option to sell the asset later to another investor who will value the asset more highly than he will.

- Investors of type a are willing to pay the following price for the asset

$$\hat{p}_a(s) = \begin{cases} \bar{p}(0) & \text{if } s_t = 0 \\ \beta(P_a(1,0)\bar{p}(0) + P_a(1,1)(1+\bar{p}(1))) & \text{if } s_t = 1 \end{cases}$$

- Investors of type b are willing to pay the following price for the asset

$$\hat{p}_b(s) = \begin{cases} \beta(P_b(0,0)\bar{p}(0) + P_b(0,1)(1+\bar{p}(1))) & \text{if } s_t = 0 \\ \bar{p}(1) & \text{if } s_t = 1 \end{cases}$$

Evidently, $\hat{p}_a(1) < \bar{p}(1)$ and $\hat{p}_b(0) < \bar{p}(0)$.

Investors of type a want to sell the asset in state 1 while investors of type b want to sell it in state 0.

- The asset changes hands whenever the state changes from 0 to 1 or from 1 to 0.
- The valuations $\hat{p}_a(s)$ and $\hat{p}_b(s)$ are displayed in the fourth and fifth rows of the table.
- Even the pessimistic investors who don't buy the asset think that it is worth more than they think future dividends are worth.

Here's code to solve for $\bar{p}$, $\hat{p}_a$ and $\hat{p}_b$ using the iterative method described above

```
In [6]: def price_optimistic_beliefs(transitions, dividend_payoff, β=.75,
                                     max_iter=50000, tol=1e-16):
            """
            Function to Solve Optimistic Beliefs
            """
            # We will guess an initial price vector of [0, 0]
            p_new = np.array([[0], [0]])
            p_old = np.array([[10.], [10.]])
```

```
    # We know this is a contraction mapping, so we can iterate to conv
    for i in range(max_iter):
        p_old = p_new
        p_new = β * np.max([q @ p_old
                            + q @ dividend_payoff for q in transitions],
                            1)

        # If we succeed in converging, break out of for loop
        if np.max(np.sqrt((p_new - p_old)**2)) < 1e-12:
            break

    ptwiddle = β * np.min([q @ p_old
                           + q @ dividend_payoff for q in transitions],
                           1)

    phat_a = np.array([p_new[0], ptwiddle[1]])
    phat_b = np.array([ptwiddle[0], p_new[1]])

    return p_new, phat_a, phat_b
```

94.4.3 Insufficient Funds

Outcomes differ when the more optimistic type of investor has insufficient wealth — or insufficient ability to borrow enough — to hold the entire stock of the asset.

In this case, the asset price must adjust to attract pessimistic investors.

Instead of equation (2), the equilibrium price satisfies

$$\check{p}(s) = \beta \min \left\{ P_a(s,1)\check{p}(0) + P_a(s,1)(1+\check{p}(1)),\ P_b(s,1)\check{p}(0) + P_b(s,1)(1+\check{p}(1)) \right\} \tag{4}$$

and the marginal investor who prices the asset is always the one that values it *less* highly than does the other type.

Now the marginal investor is always the (temporarily) pessimistic type.

Notice from the sixth row of that the pessimistic price $\underline{p}$ is lower than the homogeneous belief prices p_a and p_b in both states.

When pessimistic investors price the asset according to (4), optimistic investors think that the asset is underpriced.

If they could, optimistic investors would willingly borrow at the one-period gross interest rate β^{-1} to purchase more of the asset.

Implicit constraints on leverage prohibit them from doing so.

When optimistic investors price the asset as in equation (2), pessimistic investors think that the asset is overpriced and would like to sell the asset short.

Constraints on short sales prevent that.

Here's code to solve for $\check{p}$ using iteration

```
In [7]: def price_pessimistic_beliefs(transitions, dividend_payoff, β=.75,
                                      max_iter=50000, tol=1e-16):
        """
```

```
    Function to Solve Pessimistic Beliefs
    """
    # We will guess an initial price vector of [0, 0]
    p_new = np.array([[0], [0]])
    p_old = np.array([[10.], [10.]])

    # We know this is a contraction mapping, so we can iterate to conv
    for i in range(max_iter):
        p_old = p_new
        p_new = β * np.min([q @ p_old
                            + q @ dividend_payoff for q in transitions],
                           1)

        # If we succeed in converging, break out of for loop
        if np.max(np.sqrt((p_new - p_old)**2)) < 1e-12:
            break

    return p_new
```

94.4.4 Further Interpretation

[144] interprets the Harrison-Kreps model as a model of a bubble — a situation in which an asset price exceeds what every investor thinks is merited by the asset's underlying dividend stream.

Scheinkman stresses these features of the Harrison-Kreps model:

- Compared to the homogeneous beliefs setting leading to the pricing formula, high volume occurs when the Harrison-Kreps pricing formula prevails.

Type a investors sell the entire stock of the asset to type b investors every time the state switches from $s_t = 0$ to $s_t = 1$.

Type b investors sell the asset to type a investors every time the state switches from $s_t = 1$ to $s_t = 0$.

Scheinkman takes this as a strength of the model because he observes high volume during *famous bubbles*.

- If the *supply* of the asset is increased sufficiently either physically (more "houses" are built) or artificially (ways are invented to short sell "houses"), bubbles end when the supply has grown enough to outstrip optimistic investors' resources for purchasing the asset.
- If optimistic investors finance purchases by borrowing, tightening leverage constraints can extinguish a bubble.

Scheinkman extracts insights about the effects of financial regulations on bubbles.

He emphasizes how limiting short sales and limiting leverage have opposite effects.

94.5 Exercises

94.5.1 Exercise 1

Recreate the summary table using the functions we have built above.

s_t	0	1
p_a	1.33	1.22
p_b	1.45	1.91
p_o	1.85	2.08
p_p	1	1
$\hat{p}_a$	1.85	1.69
$\hat{p}_b$	1.69	2.08

You will first need to define the transition matrices and dividend payoff vector.

94.6 Solutions

94.6.1 Exercise 1

First, we will obtain equilibrium price vectors with homogeneous beliefs, including when all investors are optimistic or pessimistic.

In [8]:
```
qa = np.array([[1/2, 1/2], [2/3, 1/3]])    # Type a transition matrix
qb = np.array([[2/3, 1/3], [1/4, 3/4]])    # Type b transition matrix
# Optimistic investor transition matrix
qopt = np.array([[1/2, 1/2], [1/4, 3/4]])
# Pessimistic investor transition matrix
qpess = np.array([[2/3, 1/3], [2/3, 1/3]])

dividendreturn = np.array([[0], [1]])

transitions = [qa, qb, qopt, qpess]
labels = ['p_a', 'p_b', 'p_optimistic', 'p_pessimistic']

for transition, label in zip(transitions, labels):
    print(label)
    print("=" * 20)
    s0, s1 = np.round(price_single_beliefs(transition, dividendreturn), 2)
    print(f"State 0: {s0}")
    print(f"State 1: {s1}")
    print("-" * 20)
```

```
    p_a
====================
State 0: [1.33]
State 1: [1.22]
--------------------
p_b
====================
State 0: [1.45]
State 1: [1.91]
--------------------
p_optimistic
====================
State 0: [1.85]
State 1: [2.08]
--------------------
p_pessimistic
```

```
====================
State 0: [1.]
State 1: [1.]
--------------------
```

We will use the price_optimistic_beliefs function to find the price under heterogeneous beliefs.

```
In [9]: opt_beliefs = price_optimistic_beliefs([qa, qb], dividendreturn)
        labels = ['p_optimistic', 'p_hat_a', 'p_hat_b']

        for p, label in zip(opt_beliefs, labels):
            print(label)
            print("=" * 20)
            s0, s1 = np.round(p, 2)
            print(f"State 0: {s0}")
            print(f"State 1: {s1}")
            print("-" * 20)
```

```
    p_optimistic
====================
State 0: [1.85]
State 1: [2.08]
--------------------
p_hat_a
====================
State 0: [1.85]
State 1: [1.69]
--------------------
p_hat_b
====================
State 0: [1.69]
State 1: [2.08]
--------------------
```

Notice that the equilibrium price with heterogeneous beliefs is equal to the price under single beliefs with optimistic investors - this is due to the marginal investor being the temporarily optimistic type.

Footnotes

[1] By assuming that both types of agents always have "deep enough pockets" to purchase all of the asset, the model takes wealth dynamics off the table. The Harrison-Kreps model generates high trading volume when the state changes either from 0 to 1 or from 1 to 0.

Chapter 95

Two Modifications of Mean-Variance Portfolio Theory

95.1 Contents

- Overview 95.2
- Appendix 95.3

Authors: Daniel Csaba, Thomas J. Sargent and Balint Szoke

95.2 Overview

95.2.1 Remarks About Estimating Means and Variances

The famous **Black-Litterman** (1992) [26] 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:

```
In [1]: 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
```

95.2.2 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.

95.2.3 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 μ and covariance matrix Σ, which we express either as

$$\vec{r} - r_f \mathbf{1} \sim \mathcal{N}(\mu, \Sigma)$$

or

$$\vec{r} - r_f \mathbf{1} = \mu + C\epsilon$$

where $\epsilon \sim \mathcal{N}(0, I)$ is an $n \times 1$ random vector.

Let w be an $n \times 1$ vector of portfolio weights.

A portfolio consisting w earns returns

$$w'(\vec{r} - r_f \mathbf{1}) \sim \mathcal{N}(w'\mu, w'\Sigma w)$$

The **mean-variance portfolio choice problem** is to choose w to maximize

$$U(\mu, \Sigma; w) = w'\mu - \frac{\delta}{2} w'\Sigma w \tag{1}$$

where $\delta > 0$ is a risk-aversion parameter. The first-order condition for maximizing (1) with respect to the vector w is

$$\mu = \delta \Sigma w$$

which implies the following design of a risky portfolio:

$$w = (\delta \Sigma)^{-1} \mu \tag{2}$$

95.2.4 Estimating the Mean and Variance

The key inputs into the portfolio choice model (2) are

- estimates of the parameters μ, Σ of the random excess return vector$(\vec{r} - r_f \mathbf{1})$
- the risk-aversion parameter δ

A standard way of estimating μ is maximum-likelihood or least squares; that amounts to estimating μ by a sample mean of excess returns and estimating Σ by a sample covariance matrix.

95.2.5 The Black-Litterman Starting Point

When estimates of μ and Σ from historical sample means and covariances have been combined with **reasonable** values of the risk-aversion parameter δ to compute an optimal portfolio from formula (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.

In [2]:
```
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 (2) as a good model for choosing an optimal portfolio w.
- They want to continue to allow the customer to express his or her risk tolerance by setting δ.
- Leaving Σ at its maximum-likelihood value, they push μ away from its maximum value in a way designed to make portfolio choices that are more plausible in terms of conforming to what most people actually do.

In particular, given Σ and a reasonable value of δ, Black and Litterman reverse engineered a vector μ_{BL} of mean excess returns that makes the w implied by formula (2) equal the **actual** market portfolio w_m, so that

$$w_m = (\delta\Sigma)^{-1}\mu_{BL}$$

95.2.6 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 δ_m be the value of the risk aversion parameter that induces an investor to hold the market portfolio in light of the optimal portfolio choice rule (2).

Evidently, portfolio rule (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 δ_m from

- an estimate of the Sharpe-ratio, and
- our maximum likelihood estimate of σ drawn from our estimates or w_m and Σ

The second key Black-Litterman step is then to use this value of δ together with the maximum likelihood estimate of Σ to deduce a $\mu_{\mathbf{BL}}$ that verifies portfolio rule (2) at the market portfolio $w = w_m$

$$\mu_m = \delta_m\Sigma w_m$$

The starting point of the Black-Litterman portfolio choice model is thus a pair (δ_m, μ_m) that tells the customer to hold the market portfolio.

```
In [3]: # 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()
```

95.2.7 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

$$\vec{r} - r_f \mathbf{1} \sim \mathcal{N}(\mu_{BL}, \Sigma) \tag{3}$$

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 (2) to choose a portfolio.

Suppose that the customer's view is expressed by a hunch that rather than (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 $\mu_{\mathbf{BL}}$

$$\tilde{\mu} = (\Sigma^{-1} + (\tau\Sigma)^{-1})^{-1}(\Sigma^{-1}\mu_{BL} + (\tau\Sigma)^{-1}\hat{\mu}) \tag{4}$$

Black and Litterman would then advise the customer to hold the portfolio associated with these views implied by rule (2):

$$\tilde{w} = (\delta\Sigma)^{-1}\tilde{\mu}$$

This portfolio $\tilde{w}$ will deviate from the portfolio w_{BL} in amounts that depend on the mixing parameter τ.

If $\hat{\mu}$ is the maximum likelihood estimator and τ is chosen heavily to weight this view, then the customer's portfolio will involve big short-long positions.

In [4]:
```
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()
```

95.2.8 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 μ one observes the average excess returns $\hat{\mu}$ on the market according to the distribution

$$\hat{\mu} \mid \mu, \Sigma \sim \mathcal{N}(\mu, \tau\Sigma)$$

where τ is typically small capturing the idea that the variation in the mean is smaller than the variation of the individual random variable.

Given the realized excess returns one should then update the prior over the mean excess returns according to Bayes rule.

The corresponding posterior over mean excess returns is normally distributed with mean

$$(\Sigma^{-1} + (\tau\Sigma)^{-1})^{-1}(\Sigma^{-1}\mu_{BL} + (\tau\Sigma)^{-1}\hat{\mu})$$

The covariance matrix is

$$(\Sigma^{-1} + (\tau\Sigma)^{-1})^{-1}$$

Hence, the Black-Litterman recommendation is consistent with the Bayes update of the prior over the mean excess returns in light of the realized average excess returns on the market.

95.2.9 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

$$\bar{z} \cdot \bar{z} = (z - \mu)'\Sigma^{-1}(z - \mu) = \bar{d} \tag{5}$$

where $\bar{d} \in \mathbb{R}_+$ denotes the (transformation) of a particular density value.

The curves defined by equation (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) [104]).

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) [45] calls it a *curve decolletage*.

Leamer (1978) [104] 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 $\bar{d}_2$

$$\begin{aligned}\bar{d}_1(\bar{d}_2) \equiv \max_{\vec{r}_e} \quad & (\vec{r}_e - \mu_{BL})'\Sigma^{-1}(\vec{r}_e - \mu_{BL}) \\ \text{subject to} \quad & (\vec{r}_e - \hat{\mu})'(\tau\Sigma)^{-1}(\vec{r}_e - \hat{\mu}) \geq \bar{d}_2\end{aligned}$$

Denoting the multiplier on the constraint by λ, the first-order condition is

$$2(\vec{r}_e - \mu_{BL})'\Sigma^{-1} + \lambda 2(\vec{r}_e - \hat{\mu})'(\tau\Sigma)^{-1} = \mathbf{0}$$

which defines the *information contract curve* between μ_{BL} and $\hat{\mu}$

$$\vec{r}_e = (\Sigma^{-1} + \lambda(\tau\Sigma)^{-1})^{-1}(\Sigma^{-1}\mu_{BL} + \lambda(\tau\Sigma)^{-1}\hat{\mu}) \tag{6}$$

Note that if $\lambda = 1$, (6) is equivalent with (4) and it identifies one point on the information contract curve.

Furthermore, because λ is a function of the minimum likelihood $\bar{d}_2$ on the RHS of the constraint, by varying $\bar{d}_2$ (or λ), we can trace out the whole curve as the figure below illustrates.

In [5]:
```
np.random.seed(1987102)

N = 2                                                   # Number of assets
T = 200                                                 # Sample size
τ = 0.8

# Random market portfolio (sum is normalized to 1)
w_m = np.random.rand(N)
w_m = w_m / (w_m.sum())

μ = (np.random.randn(N) + 5) / 100
S = np.random.randn(N, N)
V = S @ S.T
Σ = V * (w_m @ μ)**2 / (w_m @ V @ w_m)

excess_return = stat.multivariate_normal(μ, Σ)
sample = excess_return.rvs(T)

μ_est = sample.mean(0).reshape(N, 1)
Σ_est = np.cov(sample.T)

σ_m = w_m @ Σ_est @ w_m
d_m = (w_m @ μ_est) / σ_m
μ_m = (d_m * Σ_est @ w_m).reshape(N, 1)

N_r1, N_r2 = 100, 100
r1 = np.linspace(-0.04, .1, N_r1)
r2 = np.linspace(-0.02, .15, N_r2)

λ_grid = np.linspace(.001, 20, 100)
curve = np.asarray([black_litterman(λ, μ_m, μ_est, Σ_est,
                                    τ * Σ_est).flatten() for λ in λ_grid])

λ_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()
```

Note that the line that connects the two points $\hat{\mu}$ and μ_{BL} is linear, which comes from the fact that the covariance matrices of the two competing distributions (views) are proportional to each other.

To illustrate the fact that this is not necessarily the case, consider another example using the same parameter values, except that the "second view" constituting the constraint has covariance matrix τI instead of $\tau\Sigma$.

This leads to the following figure, on which the curve connecting $\hat{\mu}$ and μ_{BL} are bending

```
In [6]: 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()
```

95.2.10 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 β_0 is the "true" value of the coefficient, then the MSE of the OLS estimator is

$$\text{mse}(\hat{\beta}_{OLS}, \beta_0) := \mathbb{E}\|\hat{\beta}_{OLS} - \beta_0\|^2 = \underbrace{\mathbb{E}\|\hat{\beta}_{OLS} - \mathbb{E}\beta_{OLS}\|^2}_{\text{variance}} + \underbrace{\|\mathbb{E}\hat{\beta}_{OLS} - \beta_0\|^2}_{\text{bias}}$$

From this decomposition, one can see that in order for the MSE to be small, both the bias and the variance terms must be small.

For example, consider the case when X is a T-vector of ones (where T is the sample size), so $\hat{\beta}_{OLS}$ is simply the sample average, while $\beta_0 \in \mathbb{R}$ is defined by the true mean of y.

In this example the MSE is

$$\text{mse}(\hat{\beta}_{OLS}, \beta_0) = \underbrace{\frac{1}{T^2}\mathbb{E}\left(\sum_{t=1}^{T}(y_t - \beta_0)\right)^2}_{\text{variance}} + \underbrace{0}_{\text{bias}}$$

However, because there is a trade-off between the estimator's bias and variance, there are cases when by permitting a small bias we can substantially reduce the variance so overall the MSE gets smaller.

A typical scenario when this proves to be useful is when the number of coefficients to be estimated is large relative to the sample size.

In these cases, one approach to handle the bias-variance trade-off is the so called *Tikhonov regularization.*

A general form with regularization matrix Γ can be written as

$$\min_{\beta} \left\{ \|X\beta - y\|^2 + \|\Gamma(\beta - \tilde{\beta})\|^2 \right\}$$

which yields the solution

$$\hat{\beta}_{Reg} = (X'X + \Gamma'\Gamma)^{-1}(X'y + \Gamma'\Gamma\tilde{\beta})$$

Substituting the value of $\hat{\beta}_{OLS}$ yields

$$\hat{\beta}_{Reg} = (X'X + \Gamma'\Gamma)^{-1}(X'X\hat{\beta}_{OLS} + \Gamma'\Gamma\tilde{\beta})$$

Often, the regularization matrix takes the form $\Gamma = \lambda I$ with $\lambda > 0$ and $\tilde{\beta} = \mathbf{0}$.

Then the Tikhonov regularization is equivalent to what is called *ridge regression* in statistics.

To illustrate how this estimator addresses the bias-variance trade-off, we compute the MSE of the ridge estimator

$$\text{mse}(\hat{\beta}_{\text{ridge}}, \beta_0) = \underbrace{\frac{1}{(T+\lambda)^2}\mathbb{E}\left(\sum_{t=1}^{T}(y_t - \beta_0)\right)^2}_{\text{variance}} + \underbrace{\left(\frac{\lambda}{T+\lambda}\right)^2 \beta_0^2}_{\text{bias}}$$

The ridge regression shrinks the coefficients of the estimated vector towards zero relative to the OLS estimates thus reducing the variance term at the cost of introducing a "small" bias.

However, there is nothing special about the zero vector.

When $\tilde{\beta} \neq \mathbf{0}$ shrinkage occurs in the direction of $\tilde{\beta}$.

Now, we can give a regularization interpretation of the Black-Litterman portfolio recommendation.

To this end, simplify first the equation (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 ι_T is a column vector of ones.

Correspondingly, the OLS regression of y on X would yield the mean excess returns as coefficients.

With $\Gamma = \sqrt{\tau T^{-1}}(I_N \otimes \iota_T)$ we can write the regularized version of the mean excess return estimation

$$\begin{aligned}
\hat{\beta}_{Reg} &= (X'X + \Gamma'\Gamma)^{-1}(X'X\hat{\beta}_{OLS} + \Gamma'\Gamma\tilde{\beta}) \\
&= (1+\tau)^{-1}X'X(X'X)^{-1}(\hat{\beta}_{OLS} + \tau\tilde{\beta}) \\
&= (1+\tau)^{-1}(\hat{\beta}_{OLS} + \tau\tilde{\beta}) \\
&= (1+\tau^{-1})^{-1}(\tau^{-1}\hat{\beta}_{OLS} + \tilde{\beta})
\end{aligned}$$

Given that $\hat{\beta}_{OLS} = \hat{\mu}$ and $\tilde{\beta} = \mu_{BL}$ in the Black-Litterman model, we have the following interpretation of the model's recommendation.

The estimated (personal) view of the mean excess returns, $\hat{\mu}$ that would lead to extreme short-long positions are "shrunk" towards the conservative market view, μ_{BL}, that leads to the more conservative market portfolio.

So the Black-Litterman procedure results in a recommendation that is a compromise between the conservative market portfolio and the more extreme portfolio that is implied by estimated "personal" views.

95.2.11 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 [76], [71] call a T operator.

Let's define the T operator as it applies to the problem at hand.

Let x be an $n \times 1$ Gaussian random vector with mean vector μ and covariance matrix $\Sigma = CC'$. This means that x can be represented as

$$x = \mu + C\epsilon$$

where $\epsilon \sim \mathcal{N}(0, I)$.

Let $\phi(\epsilon)$ denote the associated standardized Gaussian density.

Let $m(\epsilon, \mu)$ be a **likelihood ratio**, meaning that it satisfies

- $m(\epsilon, \mu) > 0$
- $\int m(\epsilon, \mu)\phi(\epsilon)d\epsilon = 1$

That is, $m(\epsilon, \mu)$ is a non-negative random variable with mean 1.

Multiplying $\phi(\epsilon)$ by the likelihood ratio $m(\epsilon, \mu)$ produces a distorted distribution for ϵ, namely

$$\tilde{\phi}(\epsilon) = m(\epsilon, \mu)\phi(\epsilon)$$

The next concept that we need is the **entropy** of the distorted distribution $\tilde{\phi}$ with respect to ϕ.

Entropy is defined as

$$\text{ent} = \int \log m(\epsilon, \mu) m(\epsilon, \mu)\phi(\epsilon) d\epsilon$$

or

$$\text{ent} = \int \log m(\epsilon, \mu)\tilde{\phi}(\epsilon) d\epsilon$$

That is, relative entropy is the expected value of the likelihood ratio m where the expectation is taken with respect to the twisted density $\tilde{\phi}$.

Relative entropy is non-negative. It is a measure of the discrepancy between two probability distributions.

As such, it plays an important role in governing the behavior of statistical tests designed to discriminate one probability distribution from another.

We are ready to define the T operator.

Let $V(x)$ be a value function.

Define

$$\begin{aligned}\mathsf{T}\left(V(x)\right) &= \min_{m(\epsilon,\mu)} \int m(\epsilon, \mu)[V(\mu + C\epsilon) + \theta \log m(\epsilon, \mu)]\phi(\epsilon) d\epsilon \\ &= -\log \theta \int \exp\left(\frac{-V(\mu + C\epsilon)}{\theta}\right) \phi(\epsilon) d\epsilon\end{aligned}$$

This asserts that T is an indirect utility function for a minimization problem in which an **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 θ is lowered, more robustness is achieved.

Note: The 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 ϵ is Gaussian with mean $v = -\theta^{-1}C'w$ and covariance matrix I (When the value function is affine, the worst-case distribution distorts the mean vector of ϵ but not the covariance matrix of ϵ).

For utility function argument $w'(\vec{r} - r_f 1)$

$$\mathsf{T}(\vec{r} - r_f \mathbf{1}) = w'\mu + \zeta - \frac{1}{2\theta} w'\Sigma w$$

and entropy is

$$\frac{v'v}{2} = \frac{1}{2\theta^2} w'CC'w$$

95.2.12 A Robust Mean-variance Portfolio Model

According to criterion (1), the mean-variance portfolio choice problem chooses w to maximize

$$E[w(\vec{r} - r_f \mathbf{1})]] - \mathrm{var}[w(\vec{r} - r_f \mathbf{1})]$$

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 μ of $\vec{r} - r_f\mathbf{1}$ with the worst-case mean

$$\mu - \theta^{-1}\Sigma w$$

Notice how the worst-case mean vector depends on the portfolio w.

The operator T is the indirect utility function that emerges from solving a problem in which an agent who chooses probabilities does so in order to minimize the expected utility of a maximizing agent (in our case, the maximizing agent chooses portfolio weights w).

The robust version of the mean-variance portfolio choice problem is then to choose a portfolio w that maximizes

$$\mathsf{T}[w(\vec{r} - r_f \mathbf{1})] - \frac{\delta}{2} w'\Sigma w$$

or

$$w'(\mu - \theta^{-1}\Sigma w) - \frac{\delta}{2} w'\Sigma w \tag{7}$$

The minimizer of (7) is

$$w_{\text{rob}} = \frac{1}{\delta + \gamma} \Sigma^{-1} \mu$$

where $\gamma \equiv \theta^{-1}$ is sometimes called the risk-sensitivity parameter.

An increase in the risk-sensitivity parameter γ shrinks the portfolio weights toward zero in the same way that an increase in risk aversion does.

95.3 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

$$\lim_{N \to \infty} \quad P\left\{ \left| S_N - \mathbb{V}X \right| > \varepsilon \right\} = 0$$

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.

95.3.1 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:

$$\frac{\text{MSE}(S_N, \sigma^2)}{\text{MSE}(\bar{X}_N, \mu)} = \frac{N2\sigma^2}{N-1} \underset{N\to\infty}{\to} 2\sigma^2$$

We are interested in how this (asymptotic) relative rate of convergence changes as increasing sampling frequency puts dependence into the data.

95.3.2 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 ∞ gives back the case of IID data.

In [7]:
```
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()
```

95.3.3 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 μ.

Again, the sample average is an unbiased estimator of the unconditional mean

$$\mathbb{E}[\bar{X}_N] = \frac{1}{N}\sum_{i=1}^{N}\mathbb{E}[X_i] = \mathbb{E}[X_0] = \mu$$

The variance of the sample mean is given by

$$\begin{aligned}
\mathbb{V}\left(\bar{X}_N\right) &= \mathbb{V}\left(\frac{1}{N}\sum_{i=1}^{N} X_i\right) \\
&= \frac{1}{N^2}\left(\sum_{i=1}^{N}\mathbb{V}(X_i) + 2\sum_{i=1}^{N-1}\sum_{s=i+1}^{N}\operatorname{cov}(X_i, X_s)\right) \\
&= \frac{1}{N^2}\left(N\gamma(0) + 2\sum_{i=1}^{N-1} i\cdot\gamma\left(h\cdot(N-i)\right)\right) \\
&= \frac{1}{N^2}\left(N\frac{\sigma^2}{2\kappa} + 2\sum_{i=1}^{N-1} i\cdot\exp(-\kappa h(N-i))\frac{\sigma^2}{2\kappa}\right)
\end{aligned}$$

It is explicit in the above equation that time dependence in the data inflates the variance of the mean estimator through the covariance terms. Moreover, as we can see, a higher sampling frequency—smaller h—makes all the covariance terms larger, everything else being fixed. This implies a relatively slower rate of convergence of the sample average for high-frequency data.

Intuitively, 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.

In [8]:
```
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
```

In [9]:
```
# 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.

Part XIV

Dynamic Programming Squared

Chapter 96

Stackelberg Plans

96.1 Contents

- Overview 96.2
- Duopoly 96.3
- The Stackelberg Problem 96.4
- Stackelberg Plan 96.5
- Recursive Representation of Stackelberg Plan 96.6
- Computing the Stackelberg Plan 96.7
- Exhibiting Time Inconsistency of Stackelberg Plan 96.8
- Recursive Formulation of the Follower's Problem 96.9
- Markov Perfect Equilibrium 96.10
- MPE vs. Stackelberg 96.11

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

96.2 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 linear optimal dynamic programming.

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:

```
In [2]: import numpy as np
        import numpy.linalg as la
        import quantecon as qe
        from quantecon import LQ
        import matplotlib.pyplot as plt
        %matplotlib inline
```

96.3 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.

96.3.1 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 $\vec{q}_2$.

96.3.2 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, γ.

This formulation captures key features of the model

- Both firms make once-and-for-all choices at time 0.
- This is true even though both firms are choosing sequences of quantities that are indexed by **time**.
- The Stackelberg leader chooses first **within time** 0, knowing that the Stackelberg follower will choose second **within time** 0.

While our abstract formulation reveals the timing protocol and equilibrium concept well, it obscures details that must be addressed when we want to compute and interpret a Stackelberg plan and the follower's best response to it.

To gain insights about these things, we study them in more detail.

96.3.3 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

$$q_{1t} + (1 + \beta + c_1) q_{1t+1} - \beta q_{1t+2} = c_0 - c_2 q_{2t+1} \tag{1}$$

Equation (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 (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

$$(1 - \frac{1 + \beta + c_1}{\beta} L + \beta^{-1} L^2) = (1 - \delta_1 L)(1 - \delta_2 L) \tag{2}$$

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 (2) with β^{-1} times this inverse operator gives the follower's decision rule for setting q_{1t+1} in the **feedback-feedforward** form.

$$q_{1t+1} = \delta_1 q_{1t} - c_0 \delta_2^{-1} \beta^{-1} \frac{1}{1 - \delta_2^{-1}} + c_2 \delta_2^{-1} \beta^{-1} \sum_{j=0}^{\infty} \delta_2^j q_{2t+j+1}, \quad t \geq 0 \tag{3}$$

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 (3) for $t \geq 0$.

We can put a sequence $\{\theta_t\}_{t=0}^{\infty}$ of Lagrange multipliers on the sequence of equations (3) and formulate the following Lagrangian for the Stackelberg leader firm 2's problem

$$\begin{aligned} \tilde{L} = & \sum_{t=0}^{\infty} \beta^t \{(a_0 - a_1(q_{1t} + q_{2t}))q_{2t} - \gamma(q_{2t+1} - q_{2t})^2\} \\ & + \sum_{t=0}^{\infty} \beta^t \theta_t \{\delta_1 q_{1t} - c_0 \delta_2^{-1} \beta^{-1} \frac{1}{1-\delta_2^{-1}} + c_2 \delta_2^{-1} \beta^{-1} \sum_{j=0}^{\infty} \delta_2^{-j} q_{2t+j+1} - q_{1t+1}\} \end{aligned} \tag{4}$$

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 (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.

96.4 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

$$-\sum_{t=0}^{\infty} \beta^t r(y_t, u_t) \tag{5}$$

The Stackelberg leader faces the model

$$\begin{bmatrix} I & 0 \\ G_{21} & G_{22} \end{bmatrix} \begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \hat{A}_{11} & \hat{A}_{12} \\ \hat{A}_{21} & \hat{A}_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + \hat{B} u_t \tag{6}$$

We assume that the matrix $\begin{bmatrix} I & 0 \\ G_{21} & G_{22} \end{bmatrix}$ on the left side of equation (6) is invertible, so that we can multiply both sides by its inverse to obtain

$$\begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + B u_t \tag{7}$$

or

$$y_{t+1} = A y_t + B u_t \tag{8}$$

96.4.1 Interpretation of the Second Block of Equations

The Stackelberg follower's best response mapping is summarized by the second block of equations of (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 (5) by choosing sequences $\{u_t, x_t, z_{t+1}\}_{t=0}^{\infty}$ subject to (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 (8) to manipulate the follower's decisions.

96.4.2 More Mechanical Details

For any vector a_t, define $\vec{a}_t = [a_t, a_{t+1} ...]$.

Define a feasible set of $(\vec{y}_1, \vec{u}_0)$ sequences

$$\Omega(y_0) = \{(\vec{y}_1, \vec{u}_0) : y_{t+1} = Ay_t + Bu_t, \forall t \geq 0\}$$

Please remember that the follower's 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.

96.4.3 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 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.

96.4.4 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

$$v(y) = \max_{u,y^*} \{-r(y,u) + \beta v(y^*)\} \tag{9}$$

where the maximization is subject to

$$y^* = Ay + Bu$$

and y^* denotes next period's value.

Substituting $v(y) = -y'Py$ into Bellman equation (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

$$x_0 = -P_{22}^{-1}P_{21}z_0$$

96.5 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.

96.5.1 Calculations to Prepare Duopoly Model

Now we'll proceed to cast our duopoly model within the framework of the more general linear-quadratic structure described above.

That will allow us to compute a Stackelberg plan simply by enlisting a Riccati equation to solve a linear-quadratic dynamic program.

As emphasized above, firm 1 acts as if firm 2's decisions $\{q_{2t+1}, v_{2t}\}_{t=0}^{\infty}$ are given and beyond its control.

96.5.2 Firm 1's Problem

We again formulate firm 1's optimum problem in terms of the Lagrangian

$$L = \sum_{t=0}^{\infty} \beta^t \{a_0 q_{1t} - a_1 q_{1t}^2 - a_1 q_{1t} q_{2t} - \gamma v_{1t}^2 + \lambda_t [q_{1t} + v_{1t} - q_{1t+1}]\}$$

Firm 1 seeks a maximum with respect to $\{q_{1t+1}, v_{1t}\}_{t=0}^{\infty}$ and a minimum with respect to $\{\lambda_t\}_{t=0}^{\infty}$.

First-order conditions for this problem are

$$\begin{aligned}
\frac{\partial L}{\partial q_{1t}} &= a_0 - 2a_1 q_{1t} - a_1 q_{2t} + \lambda_t - \beta^{-1}\lambda_{t-1} = 0, \quad t \geq 1 \\
\frac{\partial L}{\partial v_{1t}} &= -2\gamma v_{1t} + \lambda_t = 0, \quad t \geq 0
\end{aligned}$$

These first-order order conditions and the constraint $q_{1t+1} = q_{1t} + v_{1t}$ can be rearranged to take the form

$$\begin{aligned}
v_{1t} &= \beta v_{1t+1} + \frac{\beta a_0}{2\gamma} - \frac{\beta a_1}{\gamma} q_{1t+1} - \frac{\beta a_1}{2\gamma} q_{2t+1} \\
q_{t+1} &= q_{1t} + v_{1t}
\end{aligned}$$

We use these two equations as components of the following linear system that confronts a Stackelberg continuation leader at time t

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{\beta a_0}{2\gamma} & -\frac{\beta a_1}{2\gamma} & -\frac{\beta a_1}{\gamma} & \beta \end{bmatrix} \begin{bmatrix} 1 \\ q_{2t+1} \\ q_{1t+1} \\ v_{1t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \\ v_{1t} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} v_{2t}$$

Time t revenues of firm 2 are $\pi_{2t} = a_0 q_{2t} - a_1 q_{2t}^2 - a_1 q_{1t} q_{2t}$ which evidently equal

$$z_t' R_1 z_t \equiv \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \end{bmatrix}' \begin{bmatrix} 0 & \frac{a_0}{2} & 0 \\ \frac{a_0}{2} & -a_1 & -\frac{a_1}{2} \\ 0 & -\frac{a_1}{2} & 0 \end{bmatrix} \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \end{bmatrix}$$

If we set $Q = \gamma$, then firm 2's period t profits can then be written

$$y_t' R y_t - Q v_{2t}^2$$

where

$$y_t = \begin{bmatrix} z_t \\ x_t \end{bmatrix}$$

with $x_t = v_{1t}$ and

$$R = \begin{bmatrix} R_1 & 0 \\ 0 & 0 \end{bmatrix}$$

We'll report results of implementing this code soon.

But first, we want to represent the Stackelberg leader's optimal choices recursively.

It is important to do this for several reasons:

- properly to interpret a representation of the Stackelberg leader's choice as a sequence of history-dependent functions
- to formulate a recursive version of the follower's choice problem

First, let's get a recursive representation of the Stackelberg leader's choice of $\vec{q}_2$ for our duopoly model.

96.6 Recursive Representation of Stackelberg Plan

In order to attain an appropriate representation of the Stackelberg leader's history-dependent plan, we will employ what amounts to a version of the **Big K, little k** device often used in macroeconomics by distinguishing z_t, which depends partly on decisions x_t of the followers, from another vector $\check{z}_t$, which does not.

We will use $\check{z}_t$ and its history $\check{z}^t = [\check{z}_t, \check{z}_{t-1}, \dots, \check{z}_0]$ to describe the sequence of the Stackelberg leader's decisions that the Stackelberg follower takes as given.

Thus, we let $\check{y}_t' = \begin{bmatrix} \check{z}_t' & \check{x}_t' \end{bmatrix}$ with initial condition $\check{z}_0 = z_0$ given.

That we distinguish $\check{z}_t$ from z_t is part and parcel of the **Big K, little k** device in this instance.

We have demonstrated that a Stackelberg plan for $\{u_t\}_{t=0}^{\infty}$ has a recursive representation

$$\begin{aligned} \check{x}_0 &= -P_{22}^{-1} P_{21} z_0 \\ u_t &= -F \check{y}_t, \quad t \geq 0 \\ \check{y}_{t+1} &= (A - BF) \check{y}_t, \quad t \geq 0 \end{aligned}$$

From this representation, we can deduce the sequence of functions $\sigma = \{\sigma_t(\check{z}^t)\}_{t=0}^{\infty}$ that comprise a Stackelberg plan.

For convenience, let $\check{A} \equiv A - BF$ and partition $\check{A}$ conformably to the partition $y_t = \begin{bmatrix} \check{z}_t \\ \check{x}_t \end{bmatrix}$ as

$$\begin{bmatrix} \check{A}_{11} & \check{A}_{12} \\ \check{A}_{21} & \check{A}_{22} \end{bmatrix}$$

Let $H_0^0 \equiv -P_{22}^{-1}P_{21}$ so that $\check{x}_0 = H_0^0 \check{z}_0$.

Then iterations on $\check{y}_{t+1} = \check{A}\check{y}_t$ starting from initial condition $\check{y}_0 = \begin{bmatrix} \check{z}_0 \\ H_0^0 \check{z}_0 \end{bmatrix}$ imply that for $t \geq 1$

$$x_t = \sum_{j=1}^{t} H_j^t \check{z}_{t-j}$$

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

$$u_t = -F_z\check{z}_t - F_x \sum_{j=1}^{t} H_j^t z_{t-j} = \sigma_t(\check{z}^t) \tag{10}$$

Representation (10) confirms that whenever $F_x \neq 0$, the typical situation, the time t component σ_t of a Stackelberg plan is **history-dependent**, meaning that the Stackelberg leader's choice u_t depends not just on $\check{z}_t$ but on components of $\check{z}^{t-1}$.

96.6.1 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 (10) cast in terms of the history $\check{z}^t$ and **not** a corresponding representation cast in terms of z^t.

96.6.2 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.

96.6.3 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

$$\begin{bmatrix} \tilde{y}_{t+1} \\ q_{1t+1} \end{bmatrix} = \begin{bmatrix} A - BF & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \tilde{y}_t \\ q_{1t} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} x_t \tag{11}$$

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 $\vec{q}_2$.

96.6.4 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.

96.7 Computing the Stackelberg Plan

Here is our code to compute a Stackelberg plan via a linear-quadratic dynamic program as outlined above

In [3]:
```
# 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 [4]:
```
# 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]

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]]
```

96.7.1 Implied Time Series for Price and Quantities

The following code plots the price and quantities

In [5]:
```
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()
```

96.7.2 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)

In [6]:
```
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
```

In [7]:
```
# 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()
```

Out[7]:
```
True
```

In [8]:
```
# 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]
```

Out[8]: True

96.8 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 time of the time inconsistency of the Stackelberg plan

In [9]:
```
# 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]
```

In [10]:
```
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()
```

96.9 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

```
In [11]: 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]

In [12]: # 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

```
In [13]: # Maximum absolute difference in quantities over time between
         # the first and second solution methods
         np.max(np.abs(yt_tilde[4] - yt_tilde[2]))
```

```
Out[13]: 6.661338147750939e-16
```

```
In [14]: # x0 == x0_tilde
         yt[:, 0][-1] - (yt_tilde[:, 1] - yt_tilde[:, 0])[-1] < tol0
```

```
Out[14]: True
```

96.9.1 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

```
In [15]: # Policy function in the follower's problem
         F_tilde.round(4)
```

Out[15]:
```
array([[-0.    ,  0.    , -0.1032, -1.    ,  0.1032]])
```

In [16]:
```
# Value function in the Stackelberg problem
P
```

Out[16]:
```
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]])
```

In [17]:
```
# Value function in the follower's problem
P_tilde
```

Out[17]:
```
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]])
```

In [18]:
```
# 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()
```

Out[18]:
```
True
```

In [19]:
```
# 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))
```

In [20]:
```
# Value function in the follower's problem
-(y0_tilde.T @ P_tilde @ y0_tilde)[0, 0]
```

Out[20]:
```
112.65590740578058
```

In [21]:
```
# Value function with `P_guess`
-(y0_tilde.T @ P_guess @ y0_tilde)[0, 0]
```

Out[21]:
```
112.6559074057807
```

In [22]:
```
# 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")
```

In [23]:
```
# Simulate the system using `F_tilde_star` and check that it gives the
# same result as the original solution

yt_tilde_star = np.zeros((n, 5))
yt_tilde_star[0, :] = y0_tilde.flatten()

for t in range(n-1):
    yt_tilde_star[t+1, :] = (A_tilde - B_tilde @ F_tilde_star) \
        @ yt_tilde_star[t, :]

fig, ax = plt.subplots()
ax.plot(yt_tilde_star[:, 4], 'r', label="q_tilde")
ax.plot(yt_tilde[2], 'b', label="q")
ax.legend()
plt.show()
```

```
In [24]: # Maximum absolute difference
         np.max(np.abs(yt_tilde_star[:, 4] - yt_tilde[2, :-1]))
```

```
Out[24]: 0.0
```

96.10 Markov Perfect Equilibrium

The **state** vector is

$$z_t = \begin{bmatrix} 1 \\ q_{2t} \\ q_{1t} \end{bmatrix}$$

and the state transition dynamics are

$$z_{t+1} = Az_t + B_1 v_{1t} + B_2 v_{2t}$$

where A is a 3×3 identity matrix and

$$B_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad B_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

The Markov perfect decision rules are

$$v_{1t} = -F_1 z_t, \quad v_{2t} = -F_2 z_t$$

and in the Markov perfect equilibrium, the state evolves according to

$$z_{t+1} = (A - B_1 F_1 - B_2 F_2) z_t$$

In [25]:
```
# 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]]
```

In [26]:
```
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()
```

```
In [27]: # Computes the maximum difference between the two quantities of the two
 ↪firms
         np.max(np.abs(q1 - q2))
```

```
Out[27]: 6.8833827526759706e-15
```

```
In [28]: # Compute values
         u1 = (- F1 @ z).flatten()
         u2 = (- F2 @ z).flatten()

         π_1 = p * q1 - γ * (u1) ** 2
         π_2 = p * q2 - γ * (u2) ** 2

         v1_forward = np.sum(βs * π_1)
         v2_forward = np.sum(βs * π_2)

         v1_direct = (- z[:, 0].T @ P1 @ z[:, 0])
         v2_direct = (- z[:, 0].T @ P2 @ z[:, 0])

         # Display values
         print("Computed values for firm 1 and firm 2:\n")
        print(f"v1(forward sim) = {v1_forward:.4f}; v1 (direct) = {v1_direct:.4f}")
         print(f"v2 (forward sim) = {v2_forward:.4f}; v2 (direct) = {v2_direct:.
 ↪4f}")
```

```
Computed values for firm 1 and firm 2:

v1(forward sim) = 133.3303; v1 (direct) = 133.3296
v2 (forward sim) = 133.3303; v2 (direct) = 133.3296
```

In [29]:
```
# 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()
```

Out[29]: True

96.11 MPE vs. Stackelberg

In [30]:
```
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()
```

In [31]:
```
# 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
```

In [32]:
```
# Compute the difference in total value between the Stackelberg and the MPE
vt_leader[0] + vt_follower[0] - 2 * vt_MPE[0]
```

Out[32]:
```
-3.970942562087714
```

Chapter 97

Ramsey Plans, Time Inconsistency, Sustainable Plans

97.1 Contents

- Overview 97.2
- The Model 97.3
- Structure 97.4
- Intertemporal Influences 97.5
- Four Models of Government Policy 97.6
- A Ramsey Planner 97.7
- A Constrained-to-a-Constant-Growth-Rate Ramsey Government 97.8
- Markov Perfect Governments 97.9
- Equilibrium Outcomes for Three Models of Government Policy Making 97.10
- A Fourth Model of Government Decision Making 97.11
- Sustainable or Credible Plan 97.12
- Whose Credible Plan is it? 97.13
- Comparison of Equilibrium Values 97.14
- Note on Dynamic Programming Squared 97.15

Co-author: Sebastian Graves

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

97.2 Overview

This lecture describes a linear-quadratic version of a model that Guillermo Calvo [29] used to illustrate the **time inconsistency** of optimal government plans.

Like Chang [34], 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 [28], Calvo [29], Stokey [150], [151], Chari and Kehoe [35], Chang [34], and Abreu [1] as well as from chapter 19 of [108].

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:

In [2]:
```
import numpy as np
from quantecon import LQ
import matplotlib.pyplot as plt
%matplotlib inline
```

97.3 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 [28] demand function:

$$m_t - p_t = -\alpha(p_{t+1} - p_t) \ , \ \alpha > 0 \tag{1}$$

for $t \geq 0$.

Equation (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 [141] 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

$$\theta_t = \frac{\alpha}{1+\alpha}\theta_{t+1} + \frac{1}{1+\alpha}\mu_t \tag{2}$$

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 (2) can be solved forward to get:

$$\theta_t = \frac{1}{1+\alpha}\sum_{j=0}^{\infty}\left(\frac{\alpha}{1+\alpha}\right)^j \mu_{t+j} \tag{3}$$

Insight: In the spirit of Chang [34], note that equations (1) and (3) show that θ_t intermediates how choices of $\mu_{t+j},\ j = 0, 1, \ldots$ impinge on time t real balances $m_t - p_t = -\alpha\theta_t$.

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

$$x_{t+1} = Ax_t + B\mu_t \tag{4}$$

We write the model in the state-space form (4) even though θ_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 (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:

$$U(m_t - p_t) = a_0 + a_1(m_t - p_t) - \frac{a_2}{2}(m_t - p_t)^2, \quad a_0 > 0, a_1 > 0, a_2 > 0 \tag{5}$$

The "bliss level" of real balances is then $\frac{a_1}{a_2}$.

The money demand function (1) and the utility function (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 (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 μ_t.

Therefore, the one-period welfare function of a benevolent government is:

$$-s(\theta_t, \mu_t) \equiv -r(x_t, \mu_t) = \begin{bmatrix} 1 \\ \theta_t \end{bmatrix}' \begin{bmatrix} a_0 & -\frac{a_1 \alpha}{2} \\ -\frac{a_1 \alpha}{2} & -\frac{a_2 \alpha^2}{2} \end{bmatrix} \begin{bmatrix} 1 \\ \theta_t \end{bmatrix} - \frac{c}{2}\mu_t^2 = -x_t' R x_t - Q\mu_t^2 \tag{6}$$

Household welfare is summarized by:

$$v_0 = -\sum_{t=0}^{\infty} \beta^t r(x_t, \mu_t) = -\sum_{t=0}^{\infty} \beta^t s(\theta_t, \mu_t) \tag{7}$$

We can represent the dependence of v_0 on $(\vec{\theta}, \vec{\mu})$ recursively via the linear difference equation

$$v_t = -s(\theta_t, \mu_t) + \beta v_{t+1} \tag{8}$$

97.4 Structure

The following structure is induced by private agents' behavior as summarized by the demand function for money (1) that leads to equation (3) that tells how future settings of μ affect the current value of θ.

Equation (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.

97.5 Intertemporal Influences

Criterion function (7) and the constraint system (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 μ_t affects the representative household's one-period utilities at all dates $s = 0, 1, \ldots, t$.

That settings of μ 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.

97.6 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 μ_t.
- when a policymaker chooses, either at time 0 or at times $t \geq 0$.
- what a policymaker assumes about how its choice of μ_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 μ_t affects household one-period utilities at dates $s = 0, 1, \ldots, t-1$

- these two models thus employ a **Ramsey** or **Stackelberg** timing protocol.

In two other models, there is a sequence of policymakers, each of whom sets μ_t at one t only

- Each such policymaker ignores effects that its choice of μ_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 μ_t for $t = 0, 1, 2, \ldots$
  - a time t policymaker chooses μ_t only and forecasts that future government decisions are unaffected by its choice.
- A sequence of separate policymakers chooses μ_t for $t = 0, 1, 2, \ldots$
  - a time t policymaker chooses only μ_t but believes that its choice of μ_t shapes private agents' beliefs about future rates of money creation and inflation, and through them, future government actions.

97.7 A Ramsey Planner

First, we consider a Ramsey planner that chooses $\{\mu_t, \theta_t\}_{t=0}^{\infty}$ to maximize (7) subject to the law of motion (4).

We can split this problem into two stages, as in Stackelberg problems and [108] Chapter 19.

In the first stage, we take the initial inflation rate θ_0 as given, and then solve the resulting LQ dynamic programming problem.

In the second stage, we maximize over the initial inflation rate θ_0.

Define a feasible set of $(\vec{x}_1, \vec{\mu}_0)$ sequences:

$$\Omega(x_0) = \{(\vec{x}_1, \vec{\mu}_0) : x_{t+1} = Ax_t + B\mu_t \ , \ \forall t \geq 0\}$$

97.7.1 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 β.

97.7.2 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 θ_0 yields the FOC:

$$-2P_{21} - 2P_{22}\theta_0 = 0$$

which implies

$$\theta_0^* = -\frac{P_{21}}{P_{22}}$$

97.7.3 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 [34]:

$$\begin{aligned} \theta_0 &= \theta_0^* \\ \mu_t &= b_0 + b_1\theta_t \\ \theta_{t+1} &= d_0 + d_1\theta_t \end{aligned} \tag{9}$$

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 (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 (3), we obtain the same sequence $\vec{\theta}$ generated by the system (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**.

Multiple roles of θ_t

The inflation rate θ_t that appears in the system (9) and equation (3) plays three roles simultaneously:

- In equation (3), θ_t is the actual rate of inflation between t and $t+1$.
- In equation (2) and (3), θ_t is also the public's expected rate of inflation between t and $t+1$.
- In system (9), θ_t is a promised rate of inflation chosen by the Ramsey planner at time 0.

97.7.4 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

97.8 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 (2) that $\theta_t = \check{\mu}$ when the government chooses a constant μ for all $t \geq 0$.

With the quadratic form (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 μ government in order to highlight time-variation of μ_t as a telltale sign of time inconsistency of a Ramsey plan.

97.9 Markov Perfect Governments

We now change the timing protocol by considering a sequence of government policymakers, the time t representative of which chooses μ_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 μ_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 μ_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 (8) under a money supply growth rate that is forever constant at $\bar{\mu}$.

Substituting for U and θ_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 μ_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:

$$\bar{\mu} = -\frac{\alpha a_1}{\alpha^2 a_2 + (1+\alpha)c}$$

97.10 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 θ and μ 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 θ^R, μ^R and the MPE values as θ^{MPE}, μ^{MPE}.

The bliss level of inflation is denoted by θ^*.

First, we will create a class ChangLQ that solves the models and stores their values

```
In [3]: 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)
                      * 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
        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

```
In [4]: clq = ChangLQ(α=1, α0=1, α1=0.5, α2=3, c=2)
        clq.β
```

```
Out[4]: 0.8464817248906141
```

The following code generates a figure that plots the value function from the Ramsey Planner's problem, which is maximized at θ_0^R.

The figure also shows the limiting value θ_∞^R to which the inflation rate θ_t converges under the Ramsey plan and compares it to the MPE value and the bliss value.

```
In [5]: 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()

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 μ (that in turn equals an implied constant θ).

```
In [6]: 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"$\theta^\check$"]

            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 θ' chosen by a continuation Ramsey planner who inherits θ.

The right figure plots a continuation Ramsey planner's choice of μ as a function of an inherited θ.

```
In [7]: 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])
            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 μ and θ 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 μ_t to a constant for all t.

```
In [8]: 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)
```

97.10.1 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 (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 μ_t must be constant over time is time consistent (see $\check{\mu}$ and $\check{\theta}$ in the above graphs).

97.10.2 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.

97.10.3 Ramsey Plans Strike Back

Research by Abreu [1], Chari and Kehoe [35] [150], and Stokey [151] 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.

97.11 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 μ_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 μ_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 μ_t.
- at each t, the government chooses μ_t to maximize a continuation discounted utility of a representative household.

97.11.1 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 θ_0^A for $t+1$. Here $\vec{\mu}^A = \{\mu_j^A\}_{j=0}^{\infty}$ is an alternative government plan to be described below.

97.11.2 Temptation to Deviate from Plan

The government's one-period return function $s(\theta, \mu)$ described in equation (6) above has the property that for all θ

$$-s(\theta, 0) \geq -s(\theta, \mu)$$

This inequality implies that whenever the policy calls for the government to set $\mu \neq 0$, the government could raise its one-period payoff by setting $\mu = 0$.

Disappointing private sector expectations in that way would increase the government's **current** payoff but would have adverse consequences for **subsequent** government payoffs because the private sector would alter its expectations about future settings of μ.

The **temporary** gain constitutes the government's temptation to deviate from a plan.

If the government at t is to resist the temptation to raise its current payoff, it is only because it forecasts adverse consequences that its setting of μ_t would bring for continuation government payoffs via alterations in the private sector's expectations.

97.12 Sustainable or Credible Plan

We call a plan $\vec{\mu}$ **sustainable** or **credible** if at each $t \geq 0$ the government chooses to confirm private agents' prior expectation of its setting for μ_t.

The government will choose to confirm prior expectations only if the long-term **loss** from disappointing private sector expectations – coming from the government's understanding of the way the private sector adjusts its expectations in response to having its prior expectations at t disappointed – outweigh the short-term **gain** from disappointing those expectations.

The theory of sustainable or credible plans assumes throughout that private sector expectations about what future governments will do are based on the assumption that governments at times $t \geq 0$ always act to maximize the continuation discounted utilities that describe those governments' purposes.

This aspect of the theory means that credible plans always come in **pairs**:

- a credible (continuation) plan to be followed if the government at t **confirms** private sector expectations
- a credible plan to be followed if the government at t **disappoints** private sector expectations

That credible plans come in pairs threaten to bring an explosion of plans to keep track of

- each credible plan itself consists of two credible plans
- therefore, the number of plans underlying one plan is unbounded

But Dilip Abreu showed how to render manageable the number of plans that must be kept track of.

The key is an object called a **self-enforcing** plan.

97.12.1 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

$$\begin{aligned} v_j^A &= -s(\theta_j^A, \mu_j^A) + \beta v_{j+1}^A \\ &\geq -s(\theta_j^A, 0) + \beta v_0^A \equiv v_j^{A,D}, \quad j \geq 0 \end{aligned} \tag{10}$$

(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

$$\begin{aligned} v_j &= -s(\theta_j, \mu_j) + \beta v_{j+1} \\ &\geq -s(\theta_j, 0) + \beta v_0^A \quad \forall j \geq 0 \end{aligned} \tag{11}$$

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 ([1]) to construct a self-enforcing plan with low time 0 value.

97.12.2 Abreu Carrot-Stick Plan

Abreu ([1]) invented a way to create a self-enforcing plan with a low initial value.

Imitating his idea, we can construct a self-enforcing plan $\vec{\mu}$ with a low time 0 value to the government by insisting that future government decision makers set μ_t to a value yielding

low one-period utilities to the household for a long time, after which government decisions thereafter yield high one-period utilities.

- Low one-period utilities early are a **stick**
- High one-period utilities later are a **carrot**

Consider a candidate plan $\vec{\mu}^A$ that sets $\mu_t^A = \bar{\mu}$ (a high positive number) for T_A periods, and then reverts to the Ramsey plan.

Denote this sequence by $\{\mu_t^A\}_{t=0}^{\infty}$.

The sequence of inflation rates implied by this plan, $\{\theta_t^A\}_{t=0}^{\infty}$, can be calculated using:

$$\theta_t^A = \frac{1}{1+\alpha} \sum_{j=0}^{\infty} \left(\frac{\alpha}{1+\alpha}\right)^j \mu_{t+j}^A$$

The value of $\{\theta_t^A, \mu_t^A\}_{t=0}^{\infty}$ at time 0 is

$$v_0^A = -\sum_{t=0}^{T_A-1} \beta^t s(\theta_t^A, \mu_t^A) + \beta^{T_A} J(\theta_0^R)$$

For an appropriate T_A, this plan can be verified to be self-enforcing and therefore credible.

97.12.3 Example of Self-Enforcing Plan

The following example implements an Abreu stick-and-carrot plan.

The government sets $\mu_t^A = 0.1$ for $t = 0, 1, \ldots, 9$ and then starts the **Ramsey plan**.

We have computed outcomes for this plan.

For this plan, we plot the θ^A, μ^A sequences as well as the implied v^A sequence.

Notice that because the government sets money supply growth high for 10 periods, inflation starts high.

Inflation gradually slowly declines because people expect the government to lower the money growth rate after period 10.

From the 10th period onwards, the inflation rate θ_t^A associated with this **Abreu plan** starts the Ramsey plan from its beginning, i.e., $\theta_{t+10}^A = \theta_t^R \quad \forall t \geq 0$.

```
In [9]: 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.ones(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])
                        - 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)
```

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 (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

```
In [10]: np.all(clq.V_A[0:20] > clq.V_dev[0:20])
```

```
Out[10]: 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:

$$v_t^R \geq -s(\theta_t^R, 0) + \beta v_0^A, \quad \forall t \geq 0$$

```
In [11]: 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)
```

```
Out[11]: True
```

97.12.4 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:

$$\begin{aligned} \hat{\mu}_t &= \nu_\mu(v_t) \\ \theta_t &= \nu_\theta(v_t) \\ v_{t+1} &= \nu_v(v_t, \mu_t) \end{aligned} \tag{12}$$

In addition to these equations, we need an initial value v_0 to characterize a sustainable plan.

The first equation of (12) tells the recommended value of $\hat{\mu}_t$ as a function of the promised value v_t.

The second equation of (12) tells the inflation rate as a function of v_t.

The third equation of (12) updates the continuation value in a way that depends on whether the government at t confirms private agents' expectations by setting μ_t equal to the recommended value $\hat{\mu}_t$, or whether it disappoints those expectations.

97.13 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 (11) that defines a credible policy.

97.14 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 μ 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.

```
In [12]: clq.J_series[0]
```

```
Out[12]: 6.67918822960449
```

```
In [13]: clq.J_check
```

```
Out[13]: 6.676729524674898
```

```
In [14]: clq.J_MPE
```

```
Out[14]: 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.

97.15 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 (1) expresses how θ_t depends on μ_t and θ_{t+1}
- equation (4) expresses how value v_t depends on (μ_t, θ_t) and v_{t+1}

A value θ from one Bellman equation appears as an argument of a second Bellman equation for another value v.

Chapter 98

Optimal Taxation with State-Contingent Debt

98.1 Contents

- Overview 98.2
- A Competitive Equilibrium with Distorting Taxes 98.3
- Recursive Formulation of the Ramsey Problem 98.4
- Examples 98.5

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

98.2 Overview

This lecture describes a celebrated model of optimal fiscal policy by Robert E. Lucas, Jr., and Nancy Stokey [111].

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:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

98.3 A Competitive Equilibrium with Distorting Taxes

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 ℓ_t and labor n_t:

$$n_t(s^t) + \ell_t(s^t) = 1 \tag{1}$$

Output equals $n_t(s^t)$ and can be divided between $c_t(s^t)$ and $g_t(s^t)$

$$c_t(s^t) + g_t(s^t) = n_t(s^t) \tag{2}$$

A representative household's preferences over $\{c_t(s^t), \ell_t(s^t)\}_{t=0}^{\infty}$ are ordered by

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)] \tag{3}$$

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

$$g_t(s^t) = \tau_t(s^t) n_t(s^t) + \sum_{s_{t+1}} p_{t+1}(s_{t+1}|s^t) b_{t+1}(s_{t+1}|s^t) - b_t(s_t|s^{t-1}) \tag{4}$$

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

$$c_t(s^t) + \sum_{s_{t+1}} p_t(s_{t+1}|s^t) b_{t+1}(s_{t+1}|s^t) = [1 - \tau_t(s^t)]\, n_t(s^t) + b_t(s_t|s^{t-1}) \quad \forall t \geq 0 \tag{5}$$

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 (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 (3) subject to (5) and (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 (3).

98.3.1 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 $\{p_{t+1}(s_{t+1}|s^t)\}_{t=0}^{\infty}$

$$q_{t+1}^0(s^{t+1}) = p_{t+1}(s_{t+1}|s^t) q_t^0(s^t) \quad s.t. \quad q_0^0(s^0) = 1 \tag{6}$$

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.

98.3.2 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*.

1. Find the allocation that maximizes the utility of the representative household (3) subject to the feasibility constraints (1) and (2) and the implementability condition derived in step 2.

- This optimal allocation is called the **Ramsey allocation**.

1. Use the Ramsey allocation together with the formulas from step 1 to find taxes and prices.

98.3.3 The Implementability Constraint

By sequential substitution of one one-period budget constraint (5) into another, we can obtain the household's present-value budget constraint:

$$\sum_{t=0}^{\infty} \sum_{s^t} q_t^0(s^t) c_t(s^t) = \sum_{t=0}^{\infty} \sum_{s^t} q_t^0(s^t)[1 - \tau_t(s^t)] n_t(s^t) + b_0 \tag{7}$$

$\{q_t^0(s^t)\}_{t=1}^{\infty}$ 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

$$(1 - \tau_t(s^t)) = \frac{u_l(s^t)}{u_c(s^t)} \tag{8}$$

and

$$p_{t+1}(s_{t+1}|s^t) = \beta \pi(s_{t+1}|s^t) \left(\frac{u_c(s^{t+1})}{u_c(s^t)} \right) \tag{9}$$

where $\pi(s_{t+1}|s^t)$ is the probability distribution of s_{t+1} conditional on history s^t.

Equation (9) implies that the Arrow-Debreu price system satisfies

$$q_t^0(s^t) = \beta^t \pi_t(s^t) \frac{u_c(s^t)}{u_c(s^0)} \tag{10}$$

Using the first-order conditions (8) and (9) to eliminate taxes and prices from (7), we derive the *implementability condition*

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t)[u_c(s^t)c_t(s^t) - u_\ell(s^t)n_t(s^t)] - u_c(s^0)b_0 = 0 \tag{11}$$

The **Ramsey problem** is to choose a feasible allocation that maximizes

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), 1 - n_t(s^t)] \tag{12}$$

subject to (11).

98.3.4 Solution Details

First, define a "pseudo utility function"

$$V\left[c_t(s^t), n_t(s^t), \Phi\right] = u[c_t(s^t), 1 - n_t(s^t)] + \Phi\left[u_c(s^t)c_t(s^t) - u_\ell(s^t)n_t(s^t)\right] \tag{13}$$

where Φ is a Lagrange multiplier on the implementability condition (7).

Next form the Lagrangian

$$J = \sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) \Big\{ V[c_t(s^t), n_t(s^t), \Phi] + \theta_t(s^t) \Big[n_t(s^t) - c_t(s^t) - g_t(s_t) \Big] \Big\} - \Phi u_c(0) b_0 \tag{14}$$

where $\{\theta_t(s^t); \forall s^t\}_{t \geq 0}$ is a sequence of Lagrange multipliers on the feasible conditions (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 $\{\theta(s^t); \forall s^t\}_{t \geq 0}$.

The first-order conditions for the Ramsey problem for periods $t \geq 1$ and $t = 0$, respectively, are

$$\begin{aligned} c_t(s^t)\colon\ & (1+\Phi)u_c(s^t) + \Phi\left[u_{cc}(s^t)c_t(s^t) - u_{\ell c}(s^t)n_t(s^t)\right] - \theta_t(s^t) = 0, \quad t \geq 1 \\ n_t(s^t)\colon\ & -(1+\Phi)u_\ell(s^t) - \Phi\left[u_{c\ell}(s^t)c_t(s^t) - u_{\ell\ell}(s^t)n_t(s^t)\right] + \theta_t(s^t) = 0, \quad t \geq 1 \end{aligned} \tag{15}$$

and

$$
\begin{aligned}
c_0(s^0,b_0)\colon\ & (1+\Phi)u_c(s^0,b_0)+\Phi\left[u_{cc}(s^0,b_0)c_0(s^0,b_0)-u_{\ell c}(s^0,b_0)n_0(s^0,b_0)\right]-\theta_0(s^0,b_0)\\
& -\Phi u_{cc}(s^0,b_0)b_0=0\\
n_0(s^0,b_0)\colon\ & -(1+\Phi)u_\ell(s^0,b_0)-\Phi\left[u_{c\ell}(s^0,b_0)c_0(s^0,b_0)-u_{\ell\ell}(s^0,b_0)n_0(s^0,b_0)\right]+\theta_0(s^0,b_0)\\
& +\Phi u_{c\ell}(s^0,b_0)b_0=0
\end{aligned}
\tag{16}
$$

Please note how these first-order conditions differ between $t=0$ and $t\geq 1$.

It is instructive to use first-order conditions (15) for $t\geq 1$ to eliminate the multipliers $\theta_t(s^t)$.

For convenience, we suppress the time subscript and the index s^t and obtain

$$
\begin{aligned}
&(1+\Phi)u_c(c,1-c-g)+\Phi\big[cu_{cc}(c,1-c-g)-(c+g)u_{\ell c}(c,1-c-g)\big]\\
&\qquad=(1+\Phi)u_\ell(c,1-c-g)+\Phi\big[cu_{c\ell}(c,1-c-g)-(c+g)u_{\ell\ell}(c,1-c-g)\big]
\end{aligned}
\tag{17}
$$

where we have imposed conditions (1) and (2).

Equation (17) is one equation that can be solved to express the unknown c as a function of the exogenous variable g.

We also know that time $t=0$ quantities c_0 and n_0 satisfy

$$
\begin{aligned}
&(1+\Phi)u_c(c,1-c-g)+\Phi\big[cu_{cc}(c,1-c-g)-(c+g)u_{\ell c}(c,1-c-g)\big]\\
&\qquad=(1+\Phi)u_\ell(c,1-c-g)+\Phi\big[cu_{c\ell}(c,1-c-g)-(c+g)u_{\ell\ell}(c,1-c-g)\big]+\Phi(u_{cc}-u_{c,\ell})b_0
\end{aligned}
\tag{18}
$$

Notice that a counterpart to b_0 does *not* appear in (17), so c does not depend on it for $t\geq 1$.

But things are different for time $t=0$.

An analogous argument for the $t=0$ equations (16) leads to one equation that can be solved for c_0 as a function of the pair $(g(s_0),b_0)$.

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 (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.

98.3.5 The Ramsey Allocation for a Given Multiplier

Temporarily take Φ as given.

We shall compute $c_0(s^0,b_0)$ and $n_0(s^0,b_0)$ from the first-order conditions (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 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.

Φ has to take a value that assures that the household and the government's budget constraints are both satisfied at a candidate Ramsey allocation and price system associated with that Φ.

98.3.6 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 Π, where

$$\Pi(s'|s) = \text{Prob}(s_{t+1} = s'|s_t = s)$$

Also, assume that government purchases g are an exact time-invariant function $g(s)$ of s.

We maintain these assumptions throughout the remainder of this lecture.

98.3.7 Determining the Multiplier

We complete the Ramsey plan by computing the Lagrange multiplier Φ on the implementability constraint (11).

Government budget balance restricts Φ via the following line of reasoning.

The household's first-order conditions imply

$$(1 - \tau_t(s^t)) = \frac{u_l(s^t)}{u_c(s^t)} \tag{19}$$

and the implied one-period Arrow securities prices

$$p_{t+1}(s_{t+1}|s^t) = \beta \Pi(s_{t+1}|s_t) \frac{u_c(s^{t+1})}{u_c(s^t)} \tag{20}$$

Substituting from (19), (20), and the feasibility condition (2) into the recursive version (5) of the household budget constraint gives

$$\begin{aligned} u_c(s^t)[n_t(s^t) - g_t(s^t)] + \beta \sum_{s_{t+1}} \Pi(s_{t+1}|s_t) u_c(s^{t+1}) b_{t+1}(s_{t+1}|s^t) \\ = u_l(s^t) n_t(s^t) + u_c(s^t) b_t(s_t|s^{t-1}) \end{aligned} \tag{21}$$

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 (21) while β times the conditional expectation of $x_{t+1}(s^{t+1})$ appears on the left side.

Hence the equation shares much of the structure of a simple asset pricing equation with x_t being analogous to the price of the asset at time t.

We learned earlier that for a Ramsey allocation $c_t(s^t), n_t(s^t)$ and $b_t(s_t|s^{t-1})$, and therefore also $x_t(s^t)$, are each functions of s_t only, being independent of the history s^{t-1} for $t \geq 1$.

That means that we can express equation (21) as

$$u_c(s)[n(s) - g(s)] + \beta \sum_{s'} \Pi(s'|s)x'(s') = u_l(s)n(s) + x(s) \tag{22}$$

where s' denotes a next period value of s and $x'(s')$ denotes a next period value of x.

Equation (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 ith elements are the respective n, g, and x values when $s = i$, and let Π be the transition matrix for the Markov state s, then we can express (22) as the matrix equation

$$\vec{u}_c(\vec{n} - \vec{g}) + \beta\Pi\vec{x} = \vec{u}_l\vec{n} + \vec{x} \tag{23}$$

This is a system of S linear equations in the $S \times 1$ vector x, whose solution is

$$\vec{x} = (I - \beta\Pi)^{-1}[\vec{u}_c(\vec{n} - \vec{g}) - \vec{u}_l\vec{n}] \tag{24}$$

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

$$\vec{b} = \frac{\vec{x}}{\vec{u}_c} \tag{25}$$

where division here means an element-by-element division of the respective components of the $S \times 1$ vectors $\vec{x}$ and $\vec{u}_c$.

Here is a computational algorithm:

1. Start with a guess for the value for Φ, then use the first-order conditions and the feasibility conditions to compute $c(s_t), n(s_t)$ for $s \in [1, \dots, S]$ and $c_0(s_0, b_0)$ and $n_0(s_0, b_0)$, given Φ.

- these are $2(S+1)$ equations in $2(S+1)$ unknowns.

1. Solve the S equations (24) for the S elements of $\vec{x}$.

- these depend on Φ.

1. Find a Φ that satisfies

$$u_{c,0}b_0 = u_{c,0}(n_0 - g_0) - u_{l,0}n_0 + \beta \sum_{s=1}^{S} \Pi(s|s_0)x(s) \tag{26}$$

by gradually raising Φ if the left side of (26) exceeds the right side and lowering Φ if the left side is less than the right side.

1. After computing a Ramsey allocation, recover the flat tax rate on labor from (8) and the implied one-period Arrow securities prices from (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 in S components each of $\vec{c}$, $\vec{n}$, and $\vec{x}$ together with n_0, c_0, and Φ.

98.3.8 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.

98.3.9 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) [7], 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 [111] paper)

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 (17) and (18) simplify in the case of the CRRA utility function.

They become

$$
(1+\Phi)[u_c(c) + u_n(c+g)] + \Phi[cu_{cc}(c) + (c+g)u_{nn}(c+g)] = 0 \tag{27}
$$

and

$$
(1+\Phi)[u_c(c_0) + u_n(c_0+g_0)] + \Phi[c_0u_{cc}(c_0) + (c_0+g_0)u_{nn}(c_0+g_0)] - \Phi u_{cc}(c_0)b_0 = 0 \tag{28}
$$

In equation (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:

$$
b_0 + g_0 = \tau_0(c_0 + g_0) + \frac{\bar{b}}{R_0} \tag{29}
$$

where R_0 is the gross interest rate for the Markov state s_0 that is assumed to prevail at time $t = 0$ and τ_0 is the time $t = 0$ tax rate.

In equation (29), it is understood that

$$
\begin{aligned}
\tau_0 &= 1 - \frac{u_{l,0}}{u_{c,0}} \\
R_0 &= \beta \sum_{s=1}^{S} \Pi(s|s_0) \frac{u_c(s)}{u_{c,0}}
\end{aligned}
$$

98.3.10 Sequence Implementation

The above steps are implemented in a class called SequentialAllocation

```
In [3]: import numpy as np
        from scipy.optimize import root
        from quantecon import MarkovChain
```

```python
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, 0.5 * np.ones(2 * S))

        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

    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)

        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])
```

98.4 Recursive Formulation of the Ramsey Problem

$x_t(s^t) = u_c(s^t) b_t(s_t | s^{t-1})$ in equation (21) appears to be a purely "forward-looking" variable.

But $x_t(s^t)$ is a also a natural candidate for a state variable in a recursive formulation of the Ramsey problem.

98.4.1 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 times $t \geq 1$ has a different objective function and faces different constraints and state variabls 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 continuation Ramsey planner.

These lines of delegated authorities and responsibilities across time express the continuation Ramsey planners' obligations to implement their parts of the original Ramsey plan, designed once-and-for-all at time 0.

98.4.2 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 presenting a Bellman equation for $V(x, s)$ first, then a Bellman equation for $W(b, s)$.

98.4.3 The Continuation Ramsey Problem

The Bellman equation for a time $t \geq 1$ continuation Ramsey planner is

$$V(x, s) = \max_{n, \{x'(s')\}} u(n - g(s), 1 - n) + \beta \sum_{s' \in S} \Pi(s'|s) V(x', s') \tag{30}$$

where maximization over n and the S elements of $x'(s')$ is subject to the single implementability constraint for $t \geq 1$.

$$x = u_c(n - g(s)) - u_l n + \beta \sum_{s' \in S} \Pi(s'|s) x'(s') \tag{31}$$

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 (30) are $S + 1$ time-invariant policy functions

$$\begin{aligned} n_t &= f(x_t, s_t), \quad t \geq 1 \\ x_{t+1}(s_{t+1}) &= h(s_{t+1}; x_t, s_t),\ s_{t+1} \in S,\ t \geq 1 \end{aligned} \tag{32}$$

98.4.4 The Ramsey Problem

The Bellman equation for the time 0 Ramsey planner is

$$W(b_0, s_0) = \max_{n_0, \{x'(s_1)\}} u(n_0 - g_0, 1 - n_0) + \beta \sum_{s_1 \in S} \Pi(s_1|s_0) V(x'(s_1), s_1) \tag{33}$$

where maximization over n_0 and the S elements of $x'(s_1)$ is subject to the time 0 implementability constraint

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + \beta \sum_{s_1 \in S} \Pi(s_1|s_0) x'(s_1) \tag{34}$$

coming from restriction (26).

Associated with a value function $W(b_0, n_0)$ that solves Bellman equation (33) are $S+1$ time 0 policy functions

$$\begin{aligned} n_0 &= f_0(b_0, s_0) \\ x_1(s_1) &= h_0(s_1; b_0, s_0) \end{aligned} \tag{35}$$

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 (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 $E_t \sum_{\tau=t}^{\infty} \beta^{\tau - t} u(c_t, l_t)$, where the consumption and leisure processes are evaluated along the original time 0 Ramsey plan.

98.4.5 First-Order Conditions

Attach a Lagrange multiplier $\Phi_1(x, s)$ to constraint (31) and a Lagrange multiplier Φ_0 to constraint (26).

Time $t \geq 1$: the first-order conditions for the time $t \geq 1$ constrained maximization problem on the right side of the continuation Ramsey planner's Bellman equation (30) are

$$\beta \Pi(s'|s) V_x(x', s') - \beta \Pi(s'|s) \Phi_1 = 0 \tag{36}$$

for $x'(s')$ and

$$(1 + \Phi_1)(u_c - u_l) + \Phi_1 \left[n(u_{ll} - u_{lc}) + (n - g(s))(u_{cc} - u_{lc}) \right] = 0 \tag{37}$$

for n.

Given Φ_1, equation (37) is one equation to be solved for n as a function of s (or of $g(s)$).

Equation (36) implies $V_x(x', s') = \Phi_1$, while an envelope condition is $V_x(x, s) = \Phi_1$, so it follows that

$$V_x(x', s') = V_x(x, s) = \Phi_1(x, s) \tag{38}$$

Time $t = 0$: For the time 0 problem on the right side of the Ramsey planner's Bellman equation (33), first-order conditions are

$$V_x(x(s_1), s_1) = \Phi_0 \tag{39}$$

for $x(s_1), s_1 \in S$, and

$$\begin{aligned}(1 + \Phi_0)(u_{c,0} - u_{n,0}) + \Phi_0\big[n_0(u_{ll,0} - u_{lc,0}) + (n_0 - g(s_0))(u_{cc,0} - u_{cl,0})\big] \\ - \Phi_0(u_{cc,0} - u_{cl,0})b_0 = 0\end{aligned} \tag{40}$$

Notice similarities and differences between the first-order conditions for $t \geq 1$ and for $t = 0$.

An additional term is present in (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.

98.4.6 State Variable Degeneracy

Equations (39) and (40) imply that $\Phi_0 = \Phi_1$ and that

$$V_x(x_t, s_t) = \Phi_0 \tag{41}$$

for all $t \geq 1$.

When V is concave in x, this implies *state-variable degeneracy* along a Ramsey plan in the sense that for $t \geq 1$, x_t will be a time-invariant function of s_t.

Given Φ_0, this function mapping s_t into x_t can be expressed as a vector $\vec{x}$ that solves equation (34) for n and c as functions of g that are associated with $\Phi = \Phi_0$.

98.4.7 Manifestations of Time Inconsistency

While the marginal utility adjusted level of government debt x_t is a key state variable for the continuation Ramsey planners at $t \geq 1$, it is not a state variable at time 0.

The time 0 Ramsey planner faces b_0, not $x_0 = u_{c,0} b_0$, as a state variable.

The discrepancy in state variables faced by the time 0 Ramsey planner and the time $t \geq 1$ continuation Ramsey planners captures the differing obligations and incentives faced by the time 0 Ramsey planner and the time $t \geq 1$ continuation Ramsey planners.

- The time 0 Ramsey planner is obligated to honor government debt b_0 measured in time 0 consumption goods.
- The time 0 Ramsey planner can manipulate the *value* of government debt as measured by $u_{c,0} b_0$.
- In contrast, time $t \geq 1$ continuation Ramsey planners are obligated *not* to alter values of debt, as measured by $u_{c,t} b_t$, that they inherit from a preceding Ramsey planner or continuation Ramsey planner.

When government expenditures g_t are a time-invariant function of a Markov state s_t, a Ramsey plan and associated Ramsey allocation feature marginal utilities of consumption $u_c(s_t)$ that, given Φ, for $t \geq 1$ depend only on s_t, but that for $t = 0$ depend on b_0 as well.

This means that $u_c(s_t)$ will be a time-invariant function of s_t for $t \geq 1$, but except when $b_0 = 0$, a different function for $t = 0$.

This in turn means that prices of one-period Arrow securities $p_{t+1}(s_{t+1}|s_t) = p(s_{t+1}|s_t)$ will be the *same* time-invariant functions of (s_{t+1}, s_t) for $t \geq 1$, but a different function $p_0(s_1|s_0)$ for $t = 0$, except when $b_0 = 0$.

The differences between these time 0 and time $t \geq 1$ objects reflect the Ramsey planner's incentive to manipulate Arrow security prices and, through them, the value of initial government debt b_0.

98.4.8 Recursive Implementation

The above steps are implemented in a class called RecursiveAllocation

In [4]:
```
import numpy as np
from scipy.interpolate import UnivariateSpline
from scipy.optimize import fmin_slsqp
from quantecon import MarkovChain
from scipy.optimize import root


class RecursiveAllocation:

    '''
    Compute the planner's allocation by solving Bellman
    equation.
    '''

    def __init__(self, model, μgrid):

        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

        # 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
        S = len(model.π)

        # First get initial fit
        pp = SequentialAllocation(model)
        c, n, x, V = map(np.vstack, zip(*map(lambda μ: pp.time1_value(μ),
↪μgrid0)))

        Vf, cf, nf, xprimef = {}, {}, {}, {}
        for s in range(2):
            ind = np.argsort(x[:, s])   # Sort x
            # Sort arrays according to x
            c, n, x, V = c[ind], n[ind], x[ind], V[ind]
            cf[s] = UnivariateSpline(x[:, s], c[:, s])
            nf[s] = UnivariateSpline(x[:, s], n[:, s])
            Vf[s] = UnivariateSpline(x[:, s], V[:, s])
            for sprime in range(S):
                xprimef[s, sprime] = UnivariateSpline(x[:, s], x[:, s])
        policies = [cf, nf, xprimef]

        # Create xgrid
        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)
        diff = 1
        while diff > 1e-7:
            PF = T(Vf)
            Vfnew, policies = self.fit_policy_function(PF)
            diff = 0
            for s in range(S):
                diff = max(diff, np.abs(
                    (Vf[s](xgrid) - Vfnew[s](xgrid)) / Vf[s](xgrid)).max())
            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 PF using the points xgrid using
        UnivariateSpline
        '''
        xgrid, S = self.xgrid, self.S
```

```
        Vf, cf, nf, xprimef = {}, {}, {}, {}
        for s in range(S):
            PFvec = np.vstack(tuple(map(lambda x: PF(x, s), xgrid)))
            Vf[s] = UnivariateSpline(xgrid, PFvec[:, 0], s=0)
            cf[s] = UnivariateSpline(xgrid, PFvec[:, 1], s=0, k=1)
            nf[s] = UnivariateSpline(xgrid, PFvec[:, 2], s=0, k=1)
            for sprime in range(S):
                xprimef[s, sprime] = UnivariateSpline(
                    xgrid, PFvec[:, 3 + sprime], s=0, k=1)

        return Vf, [cf, nf, xprimef]


    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 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 = z0[1], z0[2], z0[3:]
        return c0, n0, xprime0


    def simulate(self, B_, s_0, T, sHist=None):
        '''
        Simulates Ramsey plan for T periods
        '''
        model, π = self.model, self.π
        Uc = model.Uc
        cf, nf, xprimef = self.policies

        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
        cHist[0], nHist[0], xprime = self.time0_allocation(B_, s_0)
        THist[0] = self.T(cHist[0], nHist[0])[s_0]
        Bhist[0] = B_
        μHist[0] = 0

        # Time 1 onward
        for t in range(1, T):
            s, x = sHist[t], xprime[sHist[t]]
            c, n, xprime = np.empty(self.S), nf[s](x), np.empty(self.S)
```

```
            for shat in range(self.S):
                c[shat] = cf[shat](x)
            for sprime in range(self.S):
                xprime[sprime] = xprimef[s, sprime](x)

            T = self.T(c, n)[s]
            u_c = Uc(c, n)
            Eu_c = π[sHist[t - 1]] @ u_c
            μHist[t] = self.Vf[s](x, 1)

            RHist[t - 1] = Uc(cHist[t - 1], nHist[t - 1]) / (self.β * Eu_c)

            cHist[t], nHist[t], Bhist[t], THist[t] = c[s], n, x / u_c[s], T

        return np.array([cHist, nHist, Bhist, THist, sHist, μHist, RHist])


class BellmanEquation:

    '''
    Bellman equation for the continuation of the Lucas-Stokey Problem
    '''

    def __init__(self, model, xgrid, policies0):

        self.β, self.π, self.G = model.β, model.π, model.G
        self.S = len(model.π)  # Number of states
        self.Θ, self.model = model.Θ, model

        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:
                xprime0 = np.empty(self.S)
                for sprime in range(self.S):
                    xprime0[sprime] = xprimef[s, sprime](x)
                self.z0[x, s] = np.hstack([cf[s](x), nf[s](x), xprime0])

        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, 0.5 * np.ones(2 * S))
        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.xFB])


    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, = self.model, self.β, self.Θ,
        G, S, π = self.G, self.S, self.π
        U, Uc, Un = model.U, model.Uc, model.Un

        def objf(z):
            c, n, xprime = z[0], z[1], z[2:]
            Vprime = np.empty(S)
            for sprime in range(S):
                Vprime[sprime] = Vf[sprime](xprime[sprime])

            return -(U(c, n) + β * π[s] @ Vprime)

        def cons(z):
            c, n, xprime = z[0], z[1], z[2:]
            return np.hstack([x - Uc(c, n) * c - Un(c, n) * n - β * π[s]
                              @ xprime,
                              (Θ * n - c - G)[s]])

        out, fx, _, imode, smode = fmin_slsqp(objf,
                                              self.z0[x, s],
                                              f_eqcons=cons,
                                              bounds=[(0, 100), (0, 100)]
                                              + [self.xbar] * S,
                                              full_output=True,
                                              iprint=0,
                                              acc=1e-10)

        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, = self.model, self.β, self.Θ,
        G, S, π = self.G, self.S, self.π
        U, Uc, Un = model.U, model.Uc, model.Un

        def objf(z):
            c, n, xprime = z[0], z[1], z[2:]
            Vprime = np.empty(S)
            for sprime in range(S):
                Vprime[sprime] = Vf[sprime](xprime[sprime])

            return -(U(c, n) + β * π[s0] @ Vprime)

        def cons(z):
            c, n, xprime = z[0], z[1], z[2:]
           return np.hstack([-Uc(c, n) * (c - B_) - Un(c, n) * n - β * π[s0]
                              @ xprime,
                              (Θ * n - c - G)[s0]])

        out, fx, _, imode, smode = fmin_slsqp(objf, self.zFB[s0],
↪f_eqcons=cons,
                                              bounds=[(0, 100), (0, 100)]
                                              + [self.xbar] * S,
                                              full_output=True, iprint=0,
                                              acc=1e-10)

        if imode > 0:
            raise Exception(smode)

        return np.hstack([-fx, out])
```

98.5 Examples

98.5.1 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

In [5]:
```
import numpy as np


class CRRAutility:

    def __init__(self,
                 β=0.9,
                 σ=2,
                 γ=2,
                 π=0.5*np.ones((2, 2)),
                 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)
```

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 $g_3 = .2$

```
In [6]: time_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]])

        time_G = np.array([0.1, 0.1, 0.1, 0.2, 0.1, 0.1])
        # Θ can in principle be random
        time_Θ = np.ones(6)
        time_example = CRRAutility(π=time_π, G=time_G, Θ=time_Θ)

        # Solve sequential problem
        time_allocation = SequentialAllocation(time_example)
        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 = time_allocation.simulate(1, 0, 7, sHist_h)
        sim_seq_l = time_allocation.simulate(1, 0, 7, sHist_l)

        # Government spending paths
        sim_seq_l[4] = time_example.G[sHist_l]
        sim_seq_h[4] = time_example.G[sHist_h]

        # Output paths
        sim_seq_l[5] = time_example.Θ[sHist_l] * sim_seq_l[1]
        sim_seq_h[5] = time_example.Θ[sHist_h] * sim_seq_h[1]

        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, sim_seq_h):
            ax.set(title=title)
            ax.plot(sim_l, '-ok', sim_h, '-or', alpha=0.7)
            ax.grid()
```

```
plt.tight_layout()
plt.show()
```

Tax smoothing

- the tax rate is constant for all $t \geq 1$
  - For $t \geq 1, t \neq 3$, this is a consequence of g_t being the same at all those dates.
  - For $t = 3$, it is a consequence of the special one-period utility function that we have assumed.
  - Under other one-period utility functions, the time $t = 3$ tax rate could be either higher or lower than for dates $t \geq 1, t \neq 3$.
- the tax rate is the same at $t = 3$ for both the high g_t outcome and the low g_t outcome

We have assumed that at $t = 0$, the government owes positive debt b_0.

It sets the time $t = 0$ tax rate partly with an eye to reducing the value $u_{c,0} b_0$ of b_0.

It does this by increasing consumption at time $t = 0$ relative to consumption in later periods.

This has the consequence of *lowering* the time $t = 0$ value of the gross interest rate for risk-free loans between periods t and $t + 1$, which equals

$$R_t = \frac{u_{c,t}}{\beta \mathbb{E}_t [u_{c,t+1}]}$$

A tax policy that makes time $t = 0$ consumption be higher than time $t = 1$ consumption evidently decreases the risk-free rate one-period interest rate, R_t, at $t = 0$.

Lowering the time $t = 0$ risk-free interest rate makes time $t = 0$ consumption goods cheaper

relative to consumption goods at later dates, thereby lowering the value $u_{c,0}b_0$ of initial government debt b_0.

We see this in a figure below that plots the time path for the risk-free interest rate under both realizations of the time $t = 3$ government expenditure shock.

The following plot illustrates how the government lowers the interest rate at time 0 by raising consumption

```
In [7]: 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()
```

98.5.2 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.

98.5.3 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.

98.5.4 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)

In [8]:
```
tax_sequence = SequentialAllocation(CRRAutility(G=0.15,
                                                π=np.ones((1, 1)),
                                                Θ=np.ones(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_sequence.simulate(gov_debt[i], 0, 2)[3]
    interest_rate[i] = tax_sequence.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()
```

The figure indicates that if the government enters with positive debt, it sets a tax rate at $t = 0$ that is less than all later tax rates.

By setting a lower tax rate at $t = 0$, the government raises consumption, which reduces the *value* $u_{c,0}b_0$ of its initial debt.

It does this by increasing c_0 and thereby lowering $u_{c,0}$.

Conversely, if $b_0 < 0$, the Ramsey planner sets the tax rate at $t = 0$ higher than in subsequent periods.

A side effect of lowering time $t = 0$ consumption is that it lowers the one-period interest rate at time $t = 0$ below that of subsequent periods.

There are only two values of initial government debt at which the tax rate is constant for all $t \geq 0$.

The first is $b_0 = 0$

- Here the government can't use the $t = 0$ tax rate to alter the value of the initial debt.

The second occurs when the government enters with sufficiently large assets that the Ramsey planner can achieve first best and sets $\tau_t = 0$ for all t.

It is only for these two values of initial government debt that the Ramsey plan is time-consistent.

Another way of saying this is that, except for these two values of initial government debt, a continuation of a Ramsey plan is not a Ramsey plan.

To illustrate this, consider a Ramsey planner who starts with an initial government debt b_1 associated with one of the Ramsey plans computed above.

Call τ_1^R the time $t = 0$ tax rate chosen by the Ramsey planner confronting this value for initial government debt government.

The figure below shows both the tax rate at time 1 chosen by our original Ramsey planner and what a new Ramsey planner would choose for its time $t = 0$ tax rate

In [9]:
```
tax_sequence = SequentialAllocation(CRRAutility(G=0.15,
                                                π=np.ones((1, 1)),
                                                Θ=np.ones(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_sequence.simulate(gov_debt[i], 0, 2)[3]
    τ_reset[i] = tax_sequence.simulate(gov_debt[i], 0, 1)[3]

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(gov_debt, tax_policy[:, 1], gov_debt, τ_reset, lw=2)
ax.set(xlabel='Initial Government Debt', title='Tax Rate',
       xlim=(min(gov_debt), max(gov_debt)))
ax.legend((r'$\tau_1$', r'$\tau_1^R$'))
ax.grid()

fig.tight_layout()
plt.show()
```

The tax rates in the figure are equal for only two values of initial government debt.

98.5.5 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

```
In [10]: import numpy as np

         class LogUtility:

             def __init__(self,
```

```
                    β=0.9,
                    ψ=0.69,
                    π=0.5*np.ones((2, 2)),
                    G=np.array([0.1, 0.2]),
                    Θ=np.ones(2),
                    transfers=False):

        self.β, self.ψ, self.π = β, ψ, π
        self.G, self.Θ, self.transfers = G, Θ, transfers

    # Utility function
    def U(self, c, n):
        return np.log(c) + self.ψ * np.log(1 - n)

    # Derivatives of utility function
    def Uc(self, c, n):
        return 1 / c

    def Ucc(self, c, n):
        return -c**(-2)

    def Un(self, c, n):
        return -self.ψ / (1 - n)

    def Unn(self, c, n):
        return -self.ψ / (1 - n)**2
```

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

```
In [11]: log_example = LogUtility()
         # Solve sequential problem
         seq_log = SequentialAllocation(log_example)

         # Initialize grid for value function iteration and solve
         μ_grid = np.linspace(-0.6, 0.0, 200)
         # Solve recursive problem
         bel_log = RecursiveAllocation(log_example, μ_grid)

         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])

         # Simulate
         sim_seq = seq_log.simulate(0.5, 0, T, sHist)
         sim_bel = bel_log.simulate(0.5, 0, T, sHist)

         # 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))
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,
↪sim_bel):
    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()
```

```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:18:
RuntimeWarning: divide by zero encountered in log
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:22:
RuntimeWarning: divide by zero encountered in double_scalars
```

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.

98.5.6 Further Comments

A related lecture describes an extension of the Lucas-Stokey model by Aiyagari, Marcet, Sargent, and Seppälä (2002) [7].

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 99

Optimal Taxation without State-Contingent Debt

99.1 Contents

- Overview 99.2
- Competitive Equilibrium with Distorting Taxes 99.3
- Recursive Version of AMSS Model 99.4
- Examples 99.5

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

99.2 Overview

Let's start with following imports:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        from scipy.optimize import root, fmin_slsqp
        from scipy.interpolate import UnivariateSpline
        from quantecon import MarkovChain
```

In an earlier lecture, we described a model of optimal taxation with state-contingent debt due to Robert E. Lucas, Jr., and Nancy Stokey [111].

Aiyagari, Marcet, Sargent, and Seppälä [7] (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.

99.3 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 ℓ_t and labor n_t:

$$n_t(s^t) + \ell_t(s^t) = 1 \tag{1}$$

Output equals $n_t(s^t)$ and can be divided between consumption $c_t(s^t)$ and $g(s_t)$

$$c_t(s^t) + g(s_t) = n_t(s^t) \tag{2}$$

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

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)] \tag{3}$$

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 [7] 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 [14] than they do in [111].

99.3.1 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 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 Section **??**

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

$$\begin{aligned} b_t(s^{t-1}) &= \tau_t^n(s^t)n_t(s^t) - g_t(s_t) - T_t(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)} \\ &\equiv z(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)}, \end{aligned} \tag{4}$$

where $z(s^t)$ 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 (4) yields:

$$b_t(s^{t-1}) = z(s^t) + \beta \sum_{s^{t+1}|s^t} \pi_{t+1}(s^{t+1}|s^t)\frac{u_c(s^{t+1})}{u_c(s^t)}\, b_{t+1}(s^t) \tag{5}$$

Components of $z(s^t)$ on the right side depend on s^t, but the left side is required to depend on s^{t-1} only.

This is what it means for one-period government debt to be risk-free.

Therefore, the sum on the right side of equation (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 (5) by the right side of next period's budget constraint (associated with a particular realization s_t) we get

$$b_t(s^{t-1}) = z(s^t) + \sum_{s^{t+1}|s^t} \beta\pi_{t+1}(s^{t+1}|s^t)\frac{u_c(s^{t+1})}{u_c(s^t)}\left[z(s^{t+1}) + \frac{b_{t+2}(s^{t+1})}{R_{t+1}(s^{t+1})}\right]$$

After making similar repeated substitutions for all future occurrences of government indebtedness, and by invoking the natural debt limit, we arrive at:

$$b_t(s^{t-1}) = \sum_{j=0}^{\infty} \sum_{s^{t+j}|s^t} \beta^j\pi_{t+j}(s^{t+j}|s^t)\frac{u_c(s^{t+j})}{u_c(s^t)}\, z(s^{t+j}) \tag{6}$$

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 $z(s^t)$ as

$$z(s^t) = \left[1 - \frac{u_\ell(s^t)}{u_c(s^t)}\right] [c_t(s^t) + g_t(s_t)] - g_t(s_t) - T_t(s^t)\,. \tag{7}$$

If we substitute the appropriate versions of the right side of (7) for $z(s^{t+j})$ into equation (6), we obtain a sequence of *implementability constraints* on a Ramsey allocation in an AMSS economy.

Expression (6) at time $t = 0$ and initial state s^0 was also an *implementability constraint* on a Ramsey allocation in a Lucas-Stokey economy:

$$b_0(s^{-1}) = \mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)}\, z(s^j) \tag{8}$$

Indeed, it was the *only* implementability constraint there.

But now we also have a large number of additional implementability constraints

$$b_t(s^{t-1}) = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)}\, z(s^{t+j}) \tag{9}$$

Equation (9) must hold for each s^t for each $t \geq 1$.

99.3.2 Comparison with Lucas-Stokey Economy

The expression on the right side of (9) in the Lucas-Stokey (1983) economy would equal the present value of a continuation stream of government 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 surpluses in the Lucas-Stokey model must belong to the **marketable subspace** of the AMSS model.

99.3.3 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

$$\max_{\{c_t(s^t), b_{t+1}(s^t)\}} \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u\left(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)\right)$$

where the maximization is subject to

$$\mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} \, z(s^j) \geq b_0(s^{-1}) \tag{10}$$

and

$$\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} \, z(s^{t+j}) = b_t(s^{t-1}) \quad \forall \, s^t \tag{11}$$

given $b_0(s^{-1})$.

Lagrangian Formulation

Let $\gamma_0(s^0)$ be a non-negative Lagrange multiplier on constraint (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 stochastic processes $\{\gamma_t(s^t)\}_{t=1}^{\infty}$ of Lagrange multipliers to the implementability constraints (11).

Depending on how the constraints bind, these multipliers can be positive or negative:

$$\begin{aligned} \gamma_t(s^t) \;\geq\; (\leq)\; 0 \quad & \text{if the constraint binds in this direction} \\ \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} \, z(s^{t+j}) \;\geq\; (\leq)\; & b_t(s^{t-1}) \end{aligned}$$

A negative multiplier $\gamma_t(s^t) < 0$ means that if we could relax constraint (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 Section **??**.

These features flow from the fact that the government cannot use state-contingent debt and therefore cannot allocate its indebtedness efficiently across future states.

99.3.4 Some Calculations

It is helpful to apply two transformations to the Lagrangian.

Multiply constraint (10) by $u_c(s^0)$ and the constraints (11) by $\beta^t u_c(s^t)$.

Then a Lagrangian for the Ramsey problem can be represented as

$$
\begin{aligned}
J &= \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \Big\{ u\left(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)\right) \\
&\qquad + \gamma_t(s^t) \Big[\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j u_c(s^{t+j})\, z(s^{t+j}) - u_c(s^t)\, b_t(s^{t-1}) \Big\} \\
&= \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \Big\{ u\left(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)\right) \\
&\qquad + \Psi_t(s^t)\, u_c(s^t)\, z(s^t) - \gamma_t(s^t)\, u_c(s^t)\, b_t(s^{t-1}) \Big\}
\end{aligned}
\tag{12}
$$

where

$$
\Psi_t(s^t) = \Psi_{t-1}(s^{t-1}) + \gamma_t(s^t) \quad \text{and} \quad \Psi_{-1}(s^{-1}) = 0 \tag{13}
$$

In (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

$$
\begin{aligned}
u_c(s^t) - u_\ell(s^t) + \Psi_t(s^t) \left\{ \left[u_{cc}(s^t) - u_{c\ell}(s^t) \right] z(s^t) + u_c(s^t)\, z_c(s^t) \right\} \\
- \gamma_t(s^t) \left[u_{cc}(s^t) - u_{c\ell}(s^t) \right] b_t(s^{t-1}) = 0
\end{aligned}
\tag{14}
$$

and with respect to $b_t(s^t)$ as

$$
\mathbb{E}_t \left[\gamma_{t+1}(s^{t+1})\, u_c(s^{t+1}) \right] = 0 \tag{15}
$$

If we substitute $z(s^t)$ from (7) and its derivative $z_c(s^t)$ into the first-order condition (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 (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.

1. The Lagrange multiplier $\Psi_t(s^t)$ in the first-order condition (14) may change over time in response to realizations of the state, while the multiplier Φ in the Lucas-Stokey economy is time-invariant.

We need some code from our an earlier lecture on optimal taxation with state-contingent debt sequential allocation implementation:

In [3]:
```
import numpy as np
from scipy.optimize import root
from quantecon import MarkovChain
```

```python
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, 0.5 * np.ones(2 * S))

        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

    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)

        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])
```

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.

99.4 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 (8) from the Lucas-Stokey economy, but
- adds measurability constraints (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.

99.4.1 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 makes a possibly important difference from AMSS [7], who restricted transfers to be non-negative).

In this case, the household faces a sequence of budget constraints

$$b_t(s^{t-1}) + (1 - \tau_t(s^t))n_t(s^t) = c_t(s^t) + b_{t+1}(s^t)/R_t(s^t) \tag{16}$$

The household's first-order conditions are $u_{c,t} = \beta R_t \mathbb{E}_t u_{c,t+1}$ and $(1-\tau_t)u_{c,t} = u_{l,t}$.

Using these to eliminate R_t and τ_t from budget constraint (16) gives

$$b_t(s^{t-1}) + \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)} n_t(s^t) = c_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1}) b_{t+1}(s^t)}{u_{c,t}(s^t)} \tag{17}$$

or

$$u_{c,t}(s^t)b_t(s^{t-1}) + u_{l,t}(s^t)n_t(s^t) = u_{c,t}(s^t)c_t(s^t) + \beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t) \tag{18}$$

Now define

$$x_t \equiv \beta b_{t+1}(s^t)\mathbb{E}_t u_{c,t+1} = u_{c,t}(s^t)\frac{b_{t+1}(s^t)}{R_t(s^t)} \tag{19}$$

and represent the household's budget constraint at time t, history s^t as

$$\frac{u_{c,t}x_{t-1}}{\beta \mathbb{E}_{t-1} u_{c,t}} = u_{c,t}c_t - u_{l,t}n_t + x_t \tag{20}$$

for $t \geq 1$.

99.4.2 Measurability Constraints

Write equation (18) as

$$b_t(s^{t-1}) = c_t(s^t) - \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)} n_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1}) b_{t+1}(s^t)}{u_{c,t}} \tag{21}$$

The right side of equation (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 (21) must equal $b_t(s^{t-1})$.

That implies that it has to be *measurable* with respect to s^{t-1}.

Equations (21) are the *measurability constraints* that the AMSS model adds to the single time 0 implementation constraint imposed in the Lucas and Stokey model.

99.4.3 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

$$V(x_-, s_-) = \max_{\{n(s), x(s)\}} \sum_s \Pi(s|s_-) \left[u(n(s) - g(s), 1 - n(s)) + \beta V(x(s), s)\right] \tag{22}$$

subject to the following collection of implementability constraints, one for each $s \in S$:

$$\frac{u_c(s)x_-}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_c(\tilde{s})} = u_c(s)(n(s) - g(s)) - u_l(s)n(s) + x(s) \tag{23}$$

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

$$W(b_0, s_0) = \max_{n_0, x_0} u(n_0 - g_0, 1 - n_0) + \beta V(x_0, s_0) \tag{24}$$

where maximization is subject to

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + x_0 \tag{25}$$

99.4.4 Martingale Supercedes State-Variable Degeneracy

Let $\mu(s|s_-)\Pi(s|s_-)$ be a Lagrange multiplier on the constraint (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

$$\beta V_x(x(s), s) = \mu(s|s_-) \tag{26}$$

Applying the envelope theorem to Bellman equation (22) gives

$$V_x(x_-, s_-) = \sum_s \Pi(s|s_-)\mu(s|s_-)\frac{u_c(s)}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s_-)u_c(\tilde{s})} \tag{27}$$

Equations (26) and (27) imply that

$$V_x(x_-, s_-) = \sum_s \left(\Pi(s|s_-)\frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}|s_-)u_c(\tilde{s})} \right) V_x(x(s), s) \tag{28}$$

Equation (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.

99.4.5 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 (i.e., x_t is an exact 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.

99.4.6 Digression on Non-negative Transfers

Throughout this lecture, we have imposed that transfers $T_t = 0$.

AMSS [7] 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 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.

99.4.7 Code

The recursive formulation is implemented as follows

```
In [4]: 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-4, tol=1e-4):

                self.β, self.π, self.G = model.β, model.π, model.G
                self.mc, self.S = MarkovChain(self.π), len(model.π)  # Number of
 ↪states
                self.Θ, self.model, self.μgrid = model.Θ, model, μgrid
                self.tol_diff, self.tol = tol_diff, tol

                # Find the first best allocation
                self.solve_time1_bellman()
                self.T.time_0 = True  # Bellman equation now solves time 0 problem

            def solve_time1_bellman(self):
                '''
                Solve the time  1 Bellman equation for calibration model and
                initial grid μgrid0
                '''
                model, μgrid0 = self.model, self.μgrid
                π = model.π
                S = len(model.π)

                # First get initial fit from Lucas Stokey solution.
                # Need to change things to be ex ante
                pp = SequentialAllocation(model)
                interp = interpolator_factory(2, None)

                def incomplete_allocation(μ_, s_):
                    c, n, x, V = pp.time1_value(μ_)
                    return c, n, π[s_] @ x, π[s_] @ V
                cf, nf, xgrid, Vf, xprimef = [], [], [], [], []
```

```
            for s_ in range(S):
                c, n, x, V = zip(*map(lambda μ: incomplete_allocation(μ, s_),
↪μgrid0))
                c, n = np.vstack(c).T, np.vstack(n).T
                x, V = np.hstack(x), np.hstack(V)
                xprimes = np.vstack([x] * S)
                cf.append(interp(x, c))
                nf.append(interp(x, n))
                Vf.append(interp(x, V))
                xgrid.append(x)
                xprimef.append(interp(x, xprimes))
        cf, nf, xprimef = fun_vstack(cf), fun_vstack(nf), fun_vstack(xprimef)
            Vf = fun_hstack(Vf)
            policies = [cf, nf, xprimef]

            # Create xgrid
            x = np.vstack(xgrid).T
            xbar = [x.min(0).max(), x.max(0).min()]
            xgrid = np.linspace(xbar[0], xbar[1], len(μgrid0))
            self.xgrid = xgrid

            # Now iterate on Bellman equation
            T = BellmanEquation(model, xgrid, policies, tol=self.tol)
            diff = 1
            while diff > self.tol_diff:
                PF = T(Vf)

                Vfnew, policies = self.fit_policy_function(PF)
                diff = np.abs((Vf(xgrid) - Vfnew(xgrid)) / Vf(xgrid)).max()

                print(diff)
                Vf = Vfnew

            # Store value function policies and Bellman Equations
            self.Vf = Vf
            self.policies = policies
            self.T = T

        def fit_policy_function(self, PF):
            '''
            Fits the policy functions
            '''
            S, xgrid = len(self.π), self.xgrid
            interp = interpolator_factory(3, 0)
            cf, nf, xprimef, Tf, Vf = [], [], [], [], []
            for s_ in range(S):
                PFvec = np.vstack([PF(x, s_) for x in self.xgrid]).T
                Vf.append(interp(xgrid, PFvec[0, :]))
                cf.append(interp(xgrid, PFvec[1:1 + S]))
                nf.append(interp(xgrid, PFvec[1 + S:1 + 2 * S]))
                xprimef.append(interp(xgrid, PFvec[1 + 2 * S:1 + 3 * S]))
                Tf.append(interp(xgrid, PFvec[1 + 3 * S:]))
            policies = fun_vstack(cf), fun_vstack(
                nf), fun_vstack(xprimef), fun_vstack(Tf)
            Vf = fun_hstack(Vf)
            return Vf, policies

        def T(self, c, n):
```

```
        '''
        Computes T given c and n
        '''
        model = self.model
        Uc, Un = model.Uc(c, n), model.Un(c, n)

        return 1 + Un / (self.Θ * Uc)

    def time0_allocation(self, B_, s0):
        '''
        Finds the optimal allocation given initial government debt B_ and
        state s_0
        '''
        PF = self.T(self.Vf)
        z0 = PF(B_, s0)
        c0, n0, xprime0, T0 = z0[1:]
        return c0, n0, xprime0, T0

    def simulate(self, B_, s_0, T, sHist=None):
        '''
        Simulates planners policies for T periods
        '''
        model, π = self.model, self.π
        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),
                                            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, 0.5 * np.ones(2 * S))
        if not res.success:
            raise Exception('Could not find first best')

        self.cFB = res.x[:S]
        self.nFB = res.x[S:]
        IFB = Uc(self.cFB, self.nFB) * self.cFB + \
            Un(self.cFB, self.nFB) * self.nFB

        self.xFB = np.linalg.solve(np.eye(S) - self.β * self.π, IFB)

        self.zFB = {}
        for s in range(S):
            self.zFB[s] = np.hstack(
                [self.cFB[s], self.nFB[s], self.π[s] @ self.xFB, 0.])

    def __call__(self, Vf):
        '''
        Given continuation value function next period return value
↪function this
        period return T(V) and optimal policies
        '''
```

```
        if not self.time_0:
            def PF(x, s): return self.get_policies_time1(x, s, Vf)
        else:
            def PF(B_, s0): return self.get_policies_time0(B_, s0, Vf)
        return PF

    def get_policies_time1(self, x, s_, Vf):
        '''
        Finds the optimal policies
        '''
        model, β, Θ, G, S, π = self.model, self.β, self.Θ, self.G, self.S,
↪self.π
        U, Uc, Un = model.U, model.Uc, model.Un

        def objf(z):
            c, n, xprime = z[:S], z[S:2 * S], z[2 * S:3 * S]

            Vprime = np.empty(S)
            for s in range(S):
                Vprime[s] = Vf[s](xprime[s])

            return -π[s_] @ (U(c, n) + β * Vprime)

        def 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,
                                              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])
```

99.5 Examples

We now turn to some examples.

We will first build some useful functions for solving the model

```
In [5]: import numpy as np
        from scipy.interpolate import UnivariateSpline


        class interpolate_wrapper:

            def __init__(self, F):
                self.F = F

            def __getitem__(self, index):
                return interpolate_wrapper(np.asarray(self.F[index]))

            def reshape(self, *args):
                self.F = self.F.reshape(*args)
                return self

            def transpose(self):
                self.F = self.F.transpose()

            def __len__(self):
                return len(self.F)

            def __call__(self, xvec):
                x = np.atleast_1d(xvec)
                shape = self.F.shape
                if len(x) == 1:
                    fhat = np.hstack([f(x) for f in self.F.flatten()])
                    return fhat.reshape(shape)
```

```
        else:
            fhat = np.vstack([f(x) for f in self.F.flatten()])
            return fhat.reshape(np.hstack((shape, len(x))))


class interpolator_factory:

    def __init__(self, k, s):
        self.k, self.s = k, s

    def __call__(self, xgrid, Fs):
        shape, m = Fs.shape[:-1], Fs.shape[-1]
        Fs = Fs.reshape((-1, m))
        F = []
        xgrid = np.sort(xgrid)  # Sort xgrid
        for Fhat in Fs:
            F.append(UnivariateSpline(xgrid, Fhat, k=self.k, s=self.s))
        return interpolate_wrapper(np.array(F).reshape(shape))


def fun_vstack(fun_list):

    Fs = [IW.F for IW in fun_list]
    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
```

99.5.1 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 consider the same government expenditure process studied in the 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.

In [6]:
```
import numpy as np


class CRRAutility:

    def __init__(self,
                 β=0.9,
                 σ=2,
                 γ=2,
                 π=0.5*np.ones((2, 2)),
                 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)
```

The following figure plots the Ramsey plan under both complete and incomplete markets for both possible realizations of the state at time $t = 3$.

Optimal policies when the government has access to state contingent debt are represented by black lines, while the optimal policies when there is only a risk-free bond are in red.

Paths with circles are histories in which there is peace, while those with triangle denote war.

```
In [7]: # Initialize μgrid for value function iteration
        μ_grid = np.linspace(-0.7, 0.01, 200)

        time_example = CRRAutility()

        time_example.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]])

        time_example.G = np.array([0.1, 0.1, 0.1, 0.2, 0.1, 0.1])
        time_example.Θ = np.ones(6)  # Θ can in principle be random

        time_example.transfers = True             # Government can use transfers
        # Solve sequential problem
        time_sequential = SequentialAllocation(time_example)
        # Solve recursive problem
        time_bellman = RecursiveAllocationAMSS(time_example, μ_grid)

        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 = time_sequential.simulate(1, 0, 7, sHist_h)
        sim_bel_h = time_bellman.simulate(1, 0, 7, sHist_h)
        sim_seq_l = time_sequential.simulate(1, 0, 7, sHist_l)
        sim_bel_l = time_bellman.simulate(1, 0, 7, sHist_l)
```

```
# Government spending paths
sim_seq_l[4] = time_example.G[sHist_l]
sim_seq_h[4] = time_example.G[sHist_h]
sim_bel_l[4] = time_example.G[sHist_l]
sim_bel_h[4] = time_example.G[sHist_h]

# Output paths
sim_seq_l[5] = time_example.Θ[sHist_l] * sim_seq_l[1]
sim_seq_h[5] = time_example.Θ[sHist_h] * sim_seq_h[1]
sim_bel_l[5] = time_example.Θ[sHist_l] * sim_bel_l[1]
sim_bel_h[5] = time_example.Θ[sHist_h] * sim_bel_h[1]


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, bel_l, bel_h in zip(axes.flatten(), titles,
                                                 sim_seq_l, sim_seq_h,
                                                 sim_bel_l, sim_bel_h):
    ax.plot(sim_l, '-ok', sim_h, '-^k', bel_l, '-or', bel_h, '-^r',
↪alpha=0.7)
    ax.set(title=title)
    ax.grid()

plt.tight_layout()
plt.show()
```

```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:24:
RuntimeWarning: divide by zero encountered in reciprocal
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:29:
RuntimeWarning: divide by zero encountered in power
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:235:
RuntimeWarning: invalid value encountered in true_divide
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:228:
RuntimeWarning: invalid value encountered in matmul
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:233:
RuntimeWarning: invalid value encountered in matmul
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:235:
RuntimeWarning: invalid value encountered in multiply
```

```
0.6029333236643755
0.11899588239403049
0.09881553212225772
0.08354106892508192
0.07149555120835548
0.06173036758118132
0.05366019901394205
0.04689112026451663
0.04115178347560931
0.036240012965927396
0.032006237992696515
0.028368464481562206
0.025192689677184087
0.022405843880195616
0.01994774715614924
0.017777614158738117
0.01586311426476452
```

```
0.014157556340393418
0.012655688350303772
0.011323561508356405
0.010134342587404501
0.009067133049314944
0.008133363039380094
0.007289176565901135
0.006541414713738157
0.005872916742002829
0.005262680193064001
0.0047307749771207785
0.00425304528362447
0.003818501528167009
0.0034264405600953744
0.003079364780532014
0.002768326786546087
0.002490427866931677
0.002240592066624134
0.0020186948255381727
0.001817134273040178
0.001636402035539666
0.0014731339707420147
0.0013228186455305523
0.0011905279885160533
0.001069923299755228
0.0009619064545164963
0.000866106560101833
0.0007801798498127538
0.0007044038334509719
0.001135820461718877
0.0005858462046557034
0.0005148785169405882
0.0008125646930954998
0.000419343630648423
0.0006110525605884945
0.0003393644339027041
0.00030505082851731526
0.0002748939327310508
0.0002466101258104514
0.00022217612526700695
0.00020017376735678401
0.00018111714263865545
0.00016358937979053516
0.00014736943218961575
0.00013236625616948046
0.00011853760872608077
0.00010958653853354627
9.594155330329376e-05
```

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 $g_3 = g_l$
- 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

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 $t = 3$.

It responds to this constraint by smoothing tax rates across time.

To finance a war it raises taxes and issues more debt.

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.

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.

In [8]:
```
import numpy as np

class LogUtility:

    def __init__(self,
                 β=0.9,
                 ψ=0.69,
                 π=0.5*np.ones((2, 2)),
                 G=np.array([0.1, 0.2]),
                 Θ=np.ones(2),
                 transfers=False):

        self.β, self.ψ, self.π = β, ψ, π
        self.G, self.Θ, self.transfers = G, Θ, transfers

    # Utility function
    def U(self, c, n):
        return np.log(c) + self.ψ * np.log(1 - n)

    # Derivatives of utility function
    def Uc(self, c, n):
        return 1 / c

    def Ucc(self, c, n):
        return -c**(-2)

    def Un(self, c, n):
        return -self.ψ / (1 - n)

    def Unn(self, c, n):
        return -self.ψ / (1 - n)**2
```

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).

In [9]:
```
log_example = LogUtility()
log_example.transfers = True                    # Government can use transfers
```

```
log_sequential = SequentialAllocation(log_example)  # Solve sequential
↪problem
log_bellman = RecursiveAllocationAMSS(log_example, μ_grid)

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])

# Simulate
sim_seq = log_sequential.simulate(0.5, 0, T, sHist)
sim_bel = log_bellman.simulate(0.5, 0, T, sHist)

titles = ['Consumption', 'Labor Supply', 'Government Debt',
          'Tax Rate', 'Government Spending', 'Output']

# Government spending paths
sim_seq[4] = log_example.G[sHist]
sim_bel[4] = log_example.G[sHist]

# Output paths
sim_seq[5] = log_example.Θ[sHist] * sim_seq[1]
sim_bel[5] = log_example.Θ[sHist] * sim_bel[1]

fig, axes = plt.subplots(3, 2, figsize=(14, 10))

for ax, title, seq, bel in zip(axes.flatten(), titles, sim_seq, sim_bel):
    ax.plot(seq, '-ok', bel, '-^b')
    ax.set(title=title)
    ax.grid()

axes[0, 0].legend(('Complete Markets', 'Incomplete Markets'))
plt.tight_layout()
plt.show()
```

```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:18:
RuntimeWarning: invalid value encountered in log
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:18:
RuntimeWarning: divide by zero encountered in log
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:22:
RuntimeWarning: divide by zero encountered in true_divide
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:235:
RuntimeWarning: invalid value encountered in true_divide
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:235:
RuntimeWarning: invalid value encountered in multiply
```

```
0.09444436241467027
0.05938723624807882
0.009418765429903522
0.008379498687574425
0.0074624123240604355
0.006647816620408291
0.005931361510280879
0.005294448322145543
0.0047253954106721
0.0042222775808757355
0.0037757367327914595
0.0033746180929005954
0.003017386825278821
```

```
0.002699930230109115
0.002417750826161132
0.002162259204654334
0.0019376221726160596
0.001735451076427532
0.0015551292357692775
0.0013916748907577743
0.0012464994947173087
0.0011179310440191763
0.0010013269547115295
0.0008961739076702308
0.0008040179931696027
0.0007206700039880485
0.0006461943981250373
0.0005794223638423901
0.0005197699346274911
0.0004655559524182191
0.00047793563176274804
0.00041453864841817386
0.0003355701386912934
0.0003008429779500316
0.00034467634902466326
0.00024187486910206502
0.0002748557784369297
0.0002832106657514851
0.00017453973560832125
0.00017016491393155364
0.00017694150942686578
0.00019907388038770387
0.00011291946575698052
0.00010121277902064972
9.094360747603131e-05
```

When the government experiences a prolonged period of peace, it is able to reduce government debt and set permanently 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.

In [10]:
```
T = 200  # 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 = ['Consumption', 'Labor Supply', 'Government Debt',
          'Tax Rate', 'Government Spending', 'Output']

# Government spending paths
sim_seq_long[4] = log_example.G[sHist_long]
sim_bel_long[4] = log_example.G[sHist_long]

# Output paths
sim_seq_long[5] = log_example.Θ[sHist_long] * sim_seq_long[1]
sim_bel_long[5] = log_example.Θ[sHist_long] * sim_bel_long[1]

fig, axes = plt.subplots(3, 2, figsize=(14, 10))

for ax, title, seq, bel in zip(axes.flatten(), titles, sim_seq_long, \
        sim_bel_long):
    ax.plot(seq, '-k', bel, '-.b', alpha=0.5)
    ax.set(title=title)
    ax.grid()

axes[0, 0].legend(('Complete Markets','Incomplete Markets'))
plt.tight_layout()
plt.show()
```

Footnotes

[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.

[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 (11) takes a positive value, so $\gamma_t(\tilde{s}^t) > 0$.

Chapter 100

Fluctuating Interest Rates Deliver Fiscal Insurance

100.1 Contents

- Overview 100.2
- Forces at Work 100.3
- Logical Flow of Lecture 100.4
- Example Economy 100.5
- Reverse Engineering Strategy 100.6
- Code for Reverse Engineering 100.7
- Short Simulation for Reverse-engineered: Initial Debt 100.8
- Long Simulation 100.9
- BEGS Approximations of Limiting Debt and Convergence Rate 100.10

Co-authors: Anmol Bhandari and David Evans

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

100.2 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 [7] model described in optimal taxation without state-contingent debt generally differs from a Ramsey allocation and Ramsey policy in the Lucas-Stokey [111] 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 measurability constraints of the AMSS model of optimal taxation without state-contingent debt are violated at the Ramsey allocation of a corresponding [111] 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 of 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 ....
  - in the tail of the Ramsey plan, the Lagrange multipliers $\gamma_t(s^t)$ on the AMSS implementability constraints (8) converge to zero.
- After the implementability constraints (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 [111] 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 purchase 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 gov-

ernment **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:

```
In [2]: import matplotlib.pyplot as plt
        %matplotlib inline
        from scipy.optimize import fsolve, fmin
```

100.3 Forces at Work

The forces driving asymptotic outcomes here are examples of dynamics present in a more general class incomplete markets models analyzed in [23] (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 the 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 rate 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 an 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.

100.4 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.

100.4.1 Equations from Lucas-Stokey (1983) Model

Although we are studying an AMSS [7] economy, a Lucas-Stokey [111] economy plays an important role in the reverse-engineering calculation to be described below.

For that reason, it is helpful to have readily available some key equations underlying a Ramsey plan for the Lucas-Stokey economy.

Recall first-order conditions for a Ramsey allocation for the Lucas-Stokey economy.

For $t \geq 1$, these take the form

$$\begin{aligned}(1+\Phi)u_c(c,1-c-g)+\Phi\big[cu_{cc}(c,1-c-g)-(c+g)u_{\ell c}(c,1-c-g)\big]\\ =(1+\Phi)u_\ell(c,1-c-g)+\Phi\big[cu_{c\ell}(c,1-c-g)-(c+g)u_{\ell\ell}(c,1-c-g)\big]\end{aligned} \tag{1}$$

There is one such equation for each value of the Markov state s_t.

In addition, given an initial Markov state, the time $t=0$ quantities c_0 and b_0 satisfy

$$\begin{aligned}(1+\Phi)u_c(c,1-c-g)+\Phi\big[cu_{cc}(c,1-c-g)-(c+g)u_{\ell c}(c,1-c-g)\big]\\ =(1+\Phi)u_\ell(c,1-c-g)+\Phi\big[cu_{c\ell}(c,1-c-g)-(c+g)u_{\ell\ell}(c,1-c-g)\big]+\Phi(u_{cc}-u_{c,\ell})b_0\end{aligned} \tag{2}$$

In addition, the time $t=0$ budget constraint is satisfied at c_0 and initial government debt b_0:

$$b_0+g_0=\tau_0(c_0+g_0)+\frac{\bar{b}}{R_0} \tag{3}$$

where R_0 is the gross interest rate for the Markov state s_0 that is assumed to prevail at time $t=0$ and τ_0 is the time $t=0$ tax rate.

In equation (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.

100.4.2 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 (1) and (2) simplify in the case of the CRRA utility function.

They become

$$(1+\Phi)[u_c(c) + u_n(c+g)] + \Phi[cu_{cc}(c) + (c+g)u_{nn}(c+g)] = 0 \tag{4}$$

and

$$(1+\Phi)[u_c(c_0) + u_n(c_0+g_0)] + \Phi[c_0 u_{cc}(c_0) + (c_0+g_0)u_{nn}(c_0+g_0)] - \Phi u_{cc}(c_0)b_0 = 0 \tag{5}$$

In equation (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.

In [3]:
```
import numpy as np

class CRRAutility:

    def __init__(self,
                 β=0.9,
                 σ=2,
                 γ=2,
                 π=0.5*np.ones((2, 2)),
                 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)
```

100.5 Example Economy

We set the following parameter values.

The Markov state s_t takes two values, namely, $0, 1$.

The initial Markov state is 0.

The Markov transition matrix is $.5I$ where I is a 2×2 identity matrix, so the s_t process is IID.

Government expenditures $g(s)$ equal .1 in Markov state 0 and .2 in Markov state 1.

We set preference parameters as follows:

$$\beta = .9$$
$$\sigma = 2$$
$$\gamma = 2$$

Here are several classes that do most of the work for us.

The code is mostly taken or adapted from the earlier lectures optimal taxation without state-contingent debt and optimal taxation with state-contingent debt.

In [4]:
```
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, 0.5 * np.ones(2 * S))

        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

    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)

        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])
```

In [5]:
```
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-4, tol=1e-4):

        self.β, self.π, self.G = model.β, model.π, model.G
        self.mc, self.S = MarkovChain(self.π), len(model.π)  # Number of
↪states
        self.Θ, self.model, self.μgrid = model.Θ, model, μgrid
        self.tol_diff, self.tol = tol_diff, tol

        # Find the first best allocation
        self.solve_time1_bellman()
        self.T.time_0 = True  # Bellman equation now solves time 0 problem

    def solve_time1_bellman(self):
```

```
        '''
        Solve the time  1 Bellman equation for calibration model and
        initial grid μgrid0
        '''
        model, μgrid0 = self.model, self.μgrid
        π = model.π
        S = len(model.π)

        # First get initial fit from Lucas Stokey solution.
        # Need to change things to be ex ante
        pp = SequentialAllocation(model)
        interp = interpolator_factory(2, None)

        def incomplete_allocation(μ_, s_):
            c, n, x, V = pp.time1_value(μ_)
            return c, n, π[s_] @ x, π[s_] @ V
        cf, nf, xgrid, Vf, xprimef = [], [], [], [], []
        for s_ in range(S):
            c, n, x, V = zip(*map(lambda μ: incomplete_allocation(μ, s_),
↪μgrid0))

            c, n = np.vstack(c).T, np.vstack(n).T
            x, V = np.hstack(x), np.hstack(V)
            xprimes = np.vstack([x] * S)
            cf.append(interp(x, c))
            nf.append(interp(x, n))
            Vf.append(interp(x, V))
            xgrid.append(x)
            xprimef.append(interp(x, xprimes))
      cf, nf, xprimef = fun_vstack(cf), fun_vstack(nf), fun_vstack(xprimef)
        Vf = fun_hstack(Vf)
        policies = [cf, nf, xprimef]

        # Create xgrid
        x = np.vstack(xgrid).T
        xbar = [x.min(0).max(), x.max(0).min()]
        xgrid = np.linspace(xbar[0], xbar[1], len(μgrid0))
        self.xgrid = xgrid

        # Now iterate on Bellman equation
        T = BellmanEquation(model, xgrid, policies, tol=self.tol)
        diff = 1
        while diff > self.tol_diff:
            PF = T(Vf)

            Vfnew, policies = self.fit_policy_function(PF)
            diff = np.abs((Vf(xgrid) - Vfnew(xgrid)) / Vf(xgrid)).max()

            print(diff)
            Vf = Vfnew

        # Store value function policies and Bellman Equations
        self.Vf = Vf
        self.policies = policies
        self.T = T

    def fit_policy_function(self, PF):
        '''
        Fits the policy functions
```

```
        '''
        S, xgrid = len(self.π), self.xgrid
        interp = interpolator_factory(3, 0)
        cf, nf, xprimef, Tf, Vf = [], [], [], [], []
        for s_ in range(S):
            PFvec = np.vstack([PF(x, s_) for x in self.xgrid]).T
            Vf.append(interp(xgrid, PFvec[0, :]))
            cf.append(interp(xgrid, PFvec[1:1 + S]))
            nf.append(interp(xgrid, PFvec[1 + S:1 + 2 * S]))
            xprimef.append(interp(xgrid, PFvec[1 + 2 * S:1 + 3 * S]))
            Tf.append(interp(xgrid, PFvec[1 + 3 * S:]))
        policies = fun_vstack(cf), fun_vstack(
            nf), fun_vstack(xprimef), fun_vstack(Tf)
        Vf = fun_hstack(Vf)
        return Vf, policies

    def T(self, c, n):
        '''
        Computes T given c and n
        '''
        model = self.model
        Uc, Un = model.Uc(c, n), model.Un(c, n)

        return 1 + Un / (self.Θ * Uc)

    def time0_allocation(self, B_, s0):
        '''
        Finds the optimal allocation given initial government debt B_ and
        state s_0
        '''
        PF = self.T(self.Vf)
        z0 = PF(B_, s0)
        c0, n0, xprime0, T0 = z0[1:]
        return c0, n0, xprime0, T0

    def simulate(self, B_, s_0, T, sHist=None):
        '''
        Simulates planners policies for T periods
        '''
        model, π = self.model, self.π
        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),
                                            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, 0.5 * np.ones(2 * S))
        if not res.success:
            raise Exception('Could not find first best')

        self.cFB = res.x[:S]
```

```
        self.nFB = res.x[S:]
        IFB = Uc(self.cFB, self.nFB) * self.cFB + \
            Un(self.cFB, self.nFB) * self.nFB

        self.xFB = np.linalg.solve(np.eye(S) - self.β * self.π, IFB)

        self.zFB = {}
        for s in range(S):
            self.zFB[s] = np.hstack(
                [self.cFB[s], self.nFB[s], self.π[s] @ self.xFB, 0.])

    def __call__(self, Vf):
        '''
        Given continuation value function next period return value function this
        period return T(V) and optimal policies
        '''
        if not self.time_0:
            def PF(x, s): return self.get_policies_time1(x, s, Vf)
        else:
            def PF(B_, s0): return self.get_policies_time0(B_, s0, Vf)
        return PF

    def get_policies_time1(self, x, s_, Vf):
        '''
        Finds the optimal policies
        '''
        model, β, Θ, G, S, π = self.model, self.β, self.Θ, self.G, self.S, self.π
        U, Uc, Un = model.U, model.Uc, model.Un

        def objf(z):
            c, n, xprime = z[:S], z[S:2 * S], z[2 * S:3 * S]

            Vprime = np.empty(S)
            for s in range(S):
                Vprime[s] = Vf[s](xprime[s])

            return -π[s_] @ (U(c, n) + β * Vprime)

        def 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,
                                              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])
```

In [6]:
```
import numpy as np
from scipy.interpolate import UnivariateSpline


class interpolate_wrapper:

    def __init__(self, F):
        self.F = F

    def __getitem__(self, index):
        return interpolate_wrapper(np.asarray(self.F[index]))

    def reshape(self, *args):
        self.F = self.F.reshape(*args)
        return self

    def transpose(self):
        self.F = self.F.transpose()
```

```
    def __len__(self):
        return len(self.F)

    def __call__(self, xvec):
        x = np.atleast_1d(xvec)
        shape = self.F.shape
        if len(x) == 1:
            fhat = np.hstack([f(x) for f in self.F.flatten()])
            return fhat.reshape(shape)
        else:
            fhat = np.vstack([f(x) for f in self.F.flatten()])
            return fhat.reshape(np.hstack((shape, len(x))))


class interpolator_factory:

    def __init__(self, k, s):
        self.k, self.s = k, s

    def __call__(self, xgrid, Fs):
        shape, m = Fs.shape[:-1], Fs.shape[-1]
        Fs = Fs.reshape((-1, m))
        F = []
        xgrid = np.sort(xgrid)  # Sort xgrid
        for Fhat in Fs:
            F.append(UnivariateSpline(xgrid, Fhat, k=self.k, s=self.s))
        return interpolate_wrapper(np.array(F).reshape(shape))


def fun_vstack(fun_list):

    Fs = [IW.F for IW in fun_list]
    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
```

100.6 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 [111] 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 [111] model presented in optimal taxation with state-contingent debt.

We use the following steps.

Step 1: Pick an initial Φ.

Step 2: Given that Φ, jointly solve two versions of equation (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 $\vec{x}$

$$\vec{x} = (I - \beta\Pi)^{-1}[\vec{u}_c(\vec{n} - \vec{g}) - \vec{u}_l\vec{n}] \tag{6}$$

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

$$\vec{b} = \frac{\vec{x}}{\vec{u}_c} \tag{7}$$

Step 5: Compute $J(\Phi) = (b(1) - b(2))^2$.

Step 6: Put steps 2 through 6 in a function minimizer and find a Φ that minimizes $J(\Phi)$.

Step 7: At the value of Φ and the value of $\bar{b}$ that emerged from step 6, solve equations (5) and (3) jointly for c_0, b_0.

100.7 Code for Reverse Engineering

Here is code to do the calculations for us.

```
In [7]: u = CRRAutility()

        def min_Φ(Φ):

            g1, g2 = u.G  # Government spending in s=0 and s=1

            # Solve Φ(c)
            def equations(unknowns, Φ):
                c1, c2 = unknowns
                # First argument of .Uc and second argument of .Un are redundant

                # Set up simultaneous equations
                eq = lambda c, g: (1 + Φ) * (u.Uc(c, 1) - -u.Un(1, c + g)) + \
                                  Φ * ((c + g) * u.Unn(1, c + g) + c * u.Ucc(c, 1))
```

```
            # Return equation evaluated at s=1 and s=2
            return np.array([eq(c1, g1), eq(c2, g2)]).flatten()

        global c1                    # Update c1 globally
        global c2                    # Update c2 globally

        c1, c2 = fsolve(equations, np.ones(2), args=(Φ))

        uc = u.Uc(np.array([c1, c2]), 1)                          # uc(n - g)
        # ul(n) = -un(c + g)
        ul = -u.Un(1, np.array([c1 + g1, c2 + g2])) * [c1 + g1, c2 + g2]
        # Solve for x
        x = np.linalg.solve(np.eye((2)) - u.β * u.π, uc * [c1, c2] - ul)

        global b                    # Update b globally
        b = x / uc
        loss = (b[0] - b[1])**2

        return loss

    Φ_star = fmin(min_Φ, .1, ftol=1e-14)
```

```
Optimization terminated successfully.
     Current function value: 0.000000
     Iterations: 24
     Function evaluations: 48
```

To recover and print out $\bar{b}$

```
In [8]: b_bar = b[0]
        b_bar
```

```
Out[8]: -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.

```
In [9]: def solve_cb(unknowns, Φ, b_bar, s=1):

            c0, b0 = unknowns

            g0 = u.G[s-1]

            R_0 = u.β * u.π[s] @ [u.Uc(c1, 1) / u.Uc(c0, 1), u.Uc(c2, 1) / u.Uc(c0,
 ↪1)]
            R_0 = 1 / R_0

            τ_0 = 1 + u.Un(1, c0 + g0) / u.Uc(c0, 1)

            eq1 = τ_0 * (c0 + g0) + b_bar / R_0 - b0 - g0
            eq2 = (1 + Φ) * (u.Uc(c0, 1)  + u.Un(1, c0 + g0)) \
                   + Φ * (c0 * u.Ucc(c0, 1) + (c0 + g0) * u.Unn(1, c0 + g0)) \
                   - Φ * u.Ucc(c0, 1) * b0

            return np.array([eq1, eq2], dtype='float64')
```

To solve the equations for c_0, b_0, we use SciPy's fsolve function

```
In [10]: c0, b0 = fsolve(solve_cb, np.array([1., -1.], dtype='float64'),
                         args=(Φ_star, b[0], 1), xtol=1.0e-12)
         c0, b0
```

```
Out[10]: (0.9344994030900681, -1.0386984075517638)
```

Thus, we have reverse engineered an initial $b0 = -1.038698407551764$ that ought to render the AMSS measurability constraints slack.

100.8 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.

```
In [11]: 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-12)

         T = 20
         sHist = np.array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
                           0, 0, 0, 1, 1, 1, 1, 1, 1, 0])


         sim_seq = log_sequential.simulate(-1.03869841, 0, T, sHist)
         sim_bel = log_bellman.simulate(-1.03869841, 0, T, sHist)

         titles = ['Consumption', 'Labor Supply', 'Government Debt',
                   'Tax Rate', 'Government Spending', 'Output']

         # Government spending paths
         sim_seq[4] = log_example.G[sHist]
         sim_bel[4] = log_example.G[sHist]

         # Output paths
         sim_seq[5] = log_example.Θ[sHist] * sim_seq[1]
         sim_bel[5] = log_example.Θ[sHist] * sim_bel[1]

         fig, axes = plt.subplots(3, 2, figsize=(14, 10))

         for ax, title, seq, bel in zip(axes.flatten(), titles, sim_seq, sim_bel):
             ax.plot(seq, '-ok', bel, '-^b')
             ax.set(title=title)
```

```
        ax.grid()

    axes[0, 0].legend(('Complete Markets', 'Incomplete Markets'))
    plt.tight_layout()
    plt.show()
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:24:
RuntimeWarning: divide by zero encountered in reciprocal
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:29:
RuntimeWarning: divide by zero encountered in power
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:235:
RuntimeWarning: invalid value encountered in true_divide
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:235:
RuntimeWarning: invalid value encountered in multiply
```

```
    0.04094445433234912
0.0016732111459338028
0.0014846748487524172
0.0013137721375787164
0.001181403713496291
0.001055965336274255
0.0009446661646844358
0.0008463807322718293
0.0007560453780620191
0.0006756001036624751
0.0006041528458700388
0.0005396004512131591
0.0004820716911559142
0.0004308273211001684
0.0003848185136981698
0.0003438352175587286
0.000307243693715206
0.0002745009148200469
0.00024531773404782317
0.0002192332430448889
0.00019593539446980383
0.00017514303514117128
0.0001565593983558638
0.00013996737141091305
0.00012514457833358872
0.00011190070779369022
0.0001000702022487836
8.949728533921615e-05
8.004975220206986e-05
7.16059059036149e-05
6.40583656889648e-05
5.731162430892402e-05
5.127968193566545e-05
4.5886529754852955e-05
4.106387898823845e-05
3.675099365037568e-05
3.289361837628717e-05
2.9443289305467077e-05
2.635678797913085e-05
2.3595484132661966e-05
2.1124903957300157e-05
1.891424711454524e-05
1.6936003234214835e-05
1.5165596593393527e-05
1.358106697950504e-05
1.2162792578343118e-05
```

```
1.089323614045592e-05
9.756722989261432e-06
8.739240835382216e-06
7.828264537526775e-06
7.012590840428639e-06
6.282206099226885e-06
5.628151985858767e-06
5.042418443402312e-06
4.5178380641774095e-06
4.048002049270609e-06
3.6271748637111453e-06
3.25022483449945e-06
2.9125597419793e-06
2.6100730258792974e-06
2.33908472396273e-06
2.096307136505147e-06
1.8787904889257265e-06
1.6838997430816734e-06
1.509274819366032e-06
1.3528011889214775e-06
1.212587081653834e-06
1.0869381104429176e-06
9.743372244174285e-07
8.73426405689756e-07
7.829877314930334e-07
7.019331006223168e-07
6.292850109121352e-07
5.641704754646274e-07
5.058062142044674e-07
4.534908905846261e-07
4.0659614636622263e-07
3.6455917260464895e-07
3.2687571576858064e-07
2.9309400626589154e-07
2.628097110920697e-07
2.3565904692627078e-07
2.1131781852307158e-07
1.894947440294367e-07
1.699288361713118e-07
1.5238586063734686e-07
1.366568424325186e-07
1.2255365279755824e-07
1.0990783200082102e-07
9.856861272368773e-08
8.840091774987147e-08
7.928334532230156e-08
7.110738489161091e-08
6.377562438179933e-08
5.720073827118772e-08
5.1304550974155735e-08
4.6016827121093976e-08
4.127508285786482e-08
3.702254013429707e-08
3.3208575403099436e-08
2.9788031505649846e-08
2.6720125194025672e-08
2.3968551794263268e-08
2.1500634727809534e-08
1.928709568259096e-08
1.7301644673193848e-08
1.5520805495718083e-08
1.3923446503682317e-08
1.2490628141347746e-08
```

```
1.1205412924843752e-08
1.005255424847768e-08
9.018420064493843e-09
8.090776959812253e-09
7.2586201295038205e-09
6.512151645666916e-09
5.842497427160883e-09
5.2417739988686235e-09
4.702866830975856e-09
4.219410867722359e-09
3.7856971691602775e-09
3.3965991981299917e-09
3.047527271191316e-09
2.73435780104547e-09
2.4533959184694e-09
2.201325576919178e-09
1.975173912964314e-09
1.7722736943474094e-09
1.5902318528480405e-09
1.4269032326934397e-09
1.280361209635549e-09
1.1488803057922307e-09
1.030910807308611e-09
9.250638131182712e-10
8.30091415855734e-10
7.44876618462649e-10
6.684152536152628e-10
5.998085081044447e-10
5.382483192957509e-10
4.830097256567513e-10
4.3344408654246964e-10
3.88969172650052e-10
3.4905943032488643e-10
3.1324806778169217e-10
2.811122777111904e-10
2.5227584505600285e-10
2.2639906361282244e-10
2.0317838832934676e-10
1.8234104590203233e-10
1.6364103618734542e-10
1.468608707188693e-10
1.3180218471597189e-10
1.182881710076278e-10
1.0616062455371046e-10
9.527750852134792e-11
```

The Ramsey allocations and Ramsey outcomes are **identical** for the Lucas-Stokey and AMSS economies.

This outcome confirms the success of our reverse-engineering exercises.

Notice how for $t \geq 1$, the tax rate is a constant - so is the par value of government debt.

However, output and labor supply are both nontrivial time-invariant functions of the Markov state.

100.9 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 afters between 1400 to 2000 periods.

For the AMSS incomplete markets economy, the marginal tax rate on labor income τ_t converges to a constant

- labor supply and output each converge to time-invariant functions of the Markov state

In [12]:
```
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()
```

100.9.1 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.

100.10 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 [23].

[23] 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 [23] expresses the time t state s government budget constraint as

$$
\mathcal{B}(s) = \mathcal{R}_\tau(s, s_-)\mathcal{B}_- + \mathcal{X}_{\tau(s)}(s) \tag{8}
$$

where the dependence on τ is to remind us that these objects depend on the tax rate and s_- is last period's Markov state.

BEGS interpret random variations in the right side of (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 $\mathcal{X}_t$

100.10.1 Asymptotic Mean

BEGS give conditions under which the ergodic mean of $\mathcal{B}_t$ is

$$\mathcal{B}^* = -\frac{\text{cov}^\infty(\mathcal{R}, \mathcal{X})}{\text{var}^\infty(\mathcal{R})} \tag{9}$$

where the superscript ∞ denotes a moment taken with respect to an ergodic distribution.

Formula (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:

$$\mathcal{B}^* = \text{argmin}_{\mathcal{B}} \text{var}(\mathcal{R}\mathcal{B} + \mathcal{X}) \tag{10}$$

The minimand in criterion (10) is the measure of fiscal risk associated with a given tax-debt policy that appears on the right side of equation (8).

Expressing formula (9) in terms of our notation tells us that $\bar{b}$ should approximately equal

$$\hat{b} = \frac{\mathcal{B}^*}{\beta E_t u_{c,t+1}} \tag{11}$$

100.10.2 Rate of Convergence

BEGS also derive the following approximation to the rate of convergence to $\mathcal{B}^*$ from an arbitrary initial condition.

$$\frac{E_t(\mathcal{B}_{t+1} - \mathcal{B}^*)}{(\mathcal{B}_t - \mathcal{B}^*)} \approx \frac{1}{1 + \beta^2 \text{var}(\mathcal{R})} \tag{12}$$

(See the equation above equation (47) in [23])

100.10.3 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 (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

$$\mu_x = \sum_s x(s)\pi(s)$$
$$\text{var}(x) = \left(\sum_s \sum_s x(s)^2 \pi(s)\right) - \mu_x^2$$
$$\text{cov}(x,y) = \left(\sum_s x(s)y(s)\pi(s)\right) - \mu_x \mu_y$$

After we compute these moments, we compute the BEGS approximation to the asymptotic mean $\hat{b}$ in formula (11).

After that, we move on to compute $\mathcal{B}^*$ in formula (9).

We'll also evaluate the BEGS criterion (8) at the limiting value $\mathcal{B}^*$

$$J(\mathcal{B}^*) = \text{var}(\mathcal{R})\left(\mathcal{B}^*\right)^2 + 2\mathcal{B}^*\text{cov}(\mathcal{R},\mathcal{X}) + \text{var}(\mathcal{X}) \tag{13}$$

Here are some functions that we'll use to compute key objects that we want

In [13]:
```
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

In [14]:
```
u = CRRAutility()

s = 0
c = [0.940580824225584, 0.8943592757759343]  # Vector for c
g = u.G       # Vector for g
n = c + g     # Total population
τ = lambda s: 1 + u.Un(1, n[s]) / u.Uc(c[s], 1)

R_s = lambda s: u.Uc(c[s], n[s]) / (u.β * (u.Uc(c[0], n[0]) * u.π[0, 0] \
                + u.Uc(c[1], n[1]) * u.π[1, 0]))
X_s = lambda s: u.Uc(c[s], n[s]) * (g[s] - τ(s) * n[s])

R = [R_s(0), R_s(1)]
X = [X_s(0), X_s(1)]

print(f"R, X = {R}, {X}")
```

```
R, X = [1.055169547122964, 1.1670526750992583], [0.06357685646224803,
0.19251010100512958]
```

Now let's compute the ingredient of the approximating limit and the approximating rate of convergence

```
In [15]: 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

Out[15]: -1.0757585378303758
```

Print out $\hat{b}$ and $\bar{b}$

```
In [16]: bhat, b_bar

Out[16]: (-1.0757585378303758, -1.0757576567504166)
```

So we have

```
In [17]: bhat - b_bar

Out[17]: -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

```
In [18]: Jmin = variance(R) * bstar**2 + 2 * bstar * covariance(R, X) + variance(X)
         Jmin

Out[18]: -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 (13) above.

Let's push our luck and compute the mean reversion speed in the formula above equation (47) in [23].

```
In [19]: 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

```
In [20]: 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.

Chapter 101

Fiscal Risk and Government Debt

101.1 Contents

- Overview 101.2
- The Economy 101.3
- Long Simulation 101.4
- Asymptotic Mean and Rate of Convergence 101.5

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install --upgrade quantecon
```

101.2 Overview

This lecture studies government debt in an AMSS economy [7] 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 applicable to more general incomplete markets economies that are presented on pages 648 - 650 in section III.D of [23] (BEGS).

We study an [7] economy with three Markov states driving government expenditures.

- In a previous lecture, we showed that with only two Markov states, it is possible that eventually endogenous interest rate fluctuations 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 where the ergodic distribution of the par value is concentrated on one point.

Nevertheless, [23] (BEGS) establish 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 a previous lecture.

Warning: Key equations in [23] section III.D carry typos that we correct below.

Let's start with some imports:

```
In [2]: import matplotlib.pyplot as plt
        %matplotlib inline
        from scipy.optimize import minimize
```

101.3 The Economy

As in Optimal Taxation without State-Contingent Debt and Optimal Taxation with State-Contingent Debt, we assume that the representative agent has utility function

$$u(c,n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma}$$

We work directly with labor supply instead of leisure.

We assume that

$$c_t + g_t = n_t$$

The Markov state s_t takes **three** values, namely, $0, 1, 2$.

The initial Markov state is 0.

The Markov transition matrix is $(1/3)I$ where I is a 3×3 identity matrix, so the s_t process is IID.

Government expenditures $g(s)$ equal .1 in Markov state 0, .2 in Markov state 1, and .3 in Markov state 2.

We set preference parameters

$$\beta = .9$$
$$\sigma = 2$$
$$\gamma = 2$$

The following Python code sets up the economy

```
In [3]: import numpy as np
```

```
class CRRAutility:

    def __init__(self,
                 β=0.9,
                 σ=2,
                 γ=2,
                 π=0.5*np.ones((2, 2)),
                 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)
```

101.3.1 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 Fiscal Insurance via Fluctuating Interest Rates.

```
In [4]: 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)
```

101.4 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

In [5]:
```
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, 0.5 * np.ones(2 * S))

        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

    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)

        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])
```

```
In [6]: 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-4, tol=1e-4):

        self.β, self.π, self.G = model.β, model.π, model.G
        self.mc, self.S = MarkovChain(self.π), len(model.π)  # Number of
↪states
        self.Θ, self.model, self.μgrid = model.Θ, model, μgrid
        self.tol_diff, self.tol = tol_diff, tol

        # Find the first best allocation
        self.solve_time1_bellman()
        self.T.time_0 = True  # Bellman equation now solves time 0 problem

    def solve_time1_bellman(self):
        '''
        Solve the time  1 Bellman equation for calibration model and
        initial grid μgrid0
        '''
        model, μgrid0 = self.model, self.μgrid
        π = model.π
        S = len(model.π)

        # First get initial fit from Lucas Stokey solution.
        # Need to change things to be ex ante
        pp = SequentialAllocation(model)
        interp = interpolator_factory(2, None)

        def incomplete_allocation(μ_, s_):
            c, n, x, V = pp.time1_value(μ_)
            return c, n, π[s_] @ x, π[s_] @ V
        cf, nf, xgrid, Vf, xprimef = [], [], [], [], []
        for s_ in range(S):
            c, n, x, V = zip(*map(lambda μ: incomplete_allocation(μ, s_),
↪μgrid0))
            c, n = np.vstack(c).T, np.vstack(n).T
            x, V = np.hstack(x), np.hstack(V)
            xprimes = np.vstack([x] * S)
            cf.append(interp(x, c))
            nf.append(interp(x, n))
            Vf.append(interp(x, V))
            xgrid.append(x)
            xprimef.append(interp(x, xprimes))
      cf, nf, xprimef = fun_vstack(cf), fun_vstack(nf), fun_vstack(xprimef)
        Vf = fun_hstack(Vf)
        policies = [cf, nf, xprimef]

        # Create xgrid
        x = np.vstack(xgrid).T
        xbar = [x.min(0).max(), x.max(0).min()]
        xgrid = np.linspace(xbar[0], xbar[1], len(μgrid0))
        self.xgrid = xgrid

        # Now iterate on Bellman equation
        T = BellmanEquation(model, xgrid, policies, tol=self.tol)
        diff = 1
        while diff > self.tol_diff:
            PF = T(Vf)
```

```
            Vfnew, policies = self.fit_policy_function(PF)
            diff = np.abs((Vf(xgrid) - Vfnew(xgrid)) / Vf(xgrid)).max()

            print(diff)
            Vf = Vfnew

        # Store value function policies and Bellman Equations
        self.Vf = Vf
        self.policies = policies
        self.T = T

    def fit_policy_function(self, PF):
        '''
        Fits the policy functions
        '''
        S, xgrid = len(self.π), self.xgrid
        interp = interpolator_factory(3, 0)
        cf, nf, xprimef, Tf, Vf = [], [], [], [], []
        for s_ in range(S):
            PFvec = np.vstack([PF(x, s_) for x in self.xgrid]).T
            Vf.append(interp(xgrid, PFvec[0, :]))
            cf.append(interp(xgrid, PFvec[1:1 + S]))
            nf.append(interp(xgrid, PFvec[1 + S:1 + 2 * S]))
            xprimef.append(interp(xgrid, PFvec[1 + 2 * S:1 + 3 * S]))
            Tf.append(interp(xgrid, PFvec[1 + 3 * S:]))
        policies = fun_vstack(cf), fun_vstack(
            nf), fun_vstack(xprimef), fun_vstack(Tf)
        Vf = fun_hstack(Vf)
        return Vf, policies

    def T(self, c, n):
        '''
        Computes T given c and n
        '''
        model = self.model
        Uc, Un = model.Uc(c, n), model.Un(c, n)

        return 1 + Un / (self.Θ * Uc)

    def time0_allocation(self, B_, s0):
        '''
        Finds the optimal allocation given initial government debt B_ and
        state s_0
        '''
        PF = self.T(self.Vf)
        z0 = PF(B_, s0)
        c0, n0, xprime0, T0 = z0[1:]
        return c0, n0, xprime0, T0

    def simulate(self, B_, s_0, T, sHist=None):
        '''
        Simulates planners policies for T periods
        '''
        model, π = self.model, self.π
        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),
                                            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, 0.5 * np.ones(2 * S))
        if not res.success:
            raise Exception('Could not find first best')

        self.cFB = res.x[:S]
        self.nFB = res.x[S:]
        IFB = Uc(self.cFB, self.nFB) * self.cFB + \
            Un(self.cFB, self.nFB) * self.nFB

        self.xFB = np.linalg.solve(np.eye(S) - self.β * self.π, IFB)

        self.zFB = {}
        for s in range(S):
            self.zFB[s] = np.hstack(
                [self.cFB[s], self.nFB[s], self.π[s] @ self.xFB, 0.])

    def __call__(self, Vf):
        '''
        Given continuation value function next period return value
↪function this
        period return T(V) and optimal policies
        '''
        if not self.time_0:
            def PF(x, s): return self.get_policies_time1(x, s, Vf)
        else:
            def PF(B_, s0): return self.get_policies_time0(B_, s0, Vf)
        return PF

    def get_policies_time1(self, x, s_, Vf):
        '''
        Finds the optimal policies
        '''
        model, β, Θ, G, S, π = self.model, self.β, self.Θ, self.G, self.S,
↪self.π
        U, Uc, Un = model.U, model.Uc, model.Un

        def objf(z):
            c, n, xprime = z[:S], z[S:2 * S], z[2 * S:3 * S]

            Vprime = np.empty(S)
            for s in range(S):
                Vprime[s] = Vf[s](xprime[s])

            return -π[s_] @ (U(c, n) + β * Vprime)

        def 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,
                                              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])
```

```
In [7]: import numpy as np
        from scipy.interpolate import UnivariateSpline
```

```python
class interpolate_wrapper:

    def __init__(self, F):
        self.F = F

    def __getitem__(self, index):
        return interpolate_wrapper(np.asarray(self.F[index]))

    def reshape(self, *args):
        self.F = self.F.reshape(*args)
        return self

    def transpose(self):
        self.F = self.F.transpose()

    def __len__(self):
        return len(self.F)

    def __call__(self, xvec):
        x = np.atleast_1d(xvec)
        shape = self.F.shape
        if len(x) == 1:
            fhat = np.hstack([f(x) for f in self.F.flatten()])
            return fhat.reshape(shape)
        else:
            fhat = np.vstack([f(x) for f in self.F.flatten()])
            return fhat.reshape(np.hstack((shape, len(x))))


class interpolator_factory:

    def __init__(self, k, s):
        self.k, self.s = k, s

    def __call__(self, xgrid, Fs):
        shape, m = Fs.shape[:-1], Fs.shape[-1]
        Fs = Fs.reshape((-1, m))
        F = []
        xgrid = np.sort(xgrid)  # Sort xgrid
        for Fhat in Fs:
            F.append(UnivariateSpline(xgrid, Fhat, k=self.k, s=self.s))
        return interpolate_wrapper(np.array(F).reshape(shape))


def fun_vstack(fun_list):

    Fs = [IW.F for IW in fun_list]
    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.

```
In [8]: μ_grid = np.linspace(-0.09, 0.1, 100)

        log_example = CRRAutility(π=(1 / 3) * np.ones((3, 3)),
                                  G=np.array([0.1, 0.2, .3]),
                                  Θ=np.ones(3))

        log_example.transfers = True        # Government can use transfers
        log_sequential = SequentialAllocation(log_example)  # Solve sequential
↪problem
        log_bellman = RecursiveAllocationAMSS(log_example, μ_grid,
                                              tol=1e-12, tol_diff=1e-10)



        T = 102000  # Set T to 102000 periods

        sim_seq_long = log_sequential.simulate(0.5, 0, T)
        sHist_long = sim_seq_long[-3]
        sim_bel_long = log_bellman.simulate(0.5, 0, T, sHist_long)

        titles = ['Government Debt', 'Tax Rate']

        fig, axes = plt.subplots(2, 1, figsize=(10, 8))

        for ax, title, id in zip(axes.flatten(), titles, [2, 3]):
            ax.plot(sim_seq_long[id], '-k', sim_bel_long[id], '-.b', alpha=0.5)
            ax.set(title=title)
            ax.grid()

        axes[0].legend(('Complete Markets', 'Incomplete Markets'))
        plt.tight_layout()
        plt.show()
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:24:
RuntimeWarning: divide by zero encountered in reciprocal
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:29:
RuntimeWarning: divide by zero encountered in power
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:235:
RuntimeWarning: invalid value encountered in true_divide
/home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:235:
RuntimeWarning: invalid value encountered in multiply
```

```
    0.03826635338765925
0.0015144378246584984
```

```
0.0013387575049829455
0.0011833202399953704
0.0010600307116151308
0.0009506620325028087
0.0008518776516937746
0.0007625857030716029
0.0006819563061621401
0.0006094002926927259
0.0005443007358227137
0.0004859950035124384
0.00043383959352032413
0.00038722730861434493
0.000345595412214899
0.0003084287064063272
0.0002752590187094664
0.0002456631291600592
0.00021925988530998263
0.00019570695817042554
0.00017469751640521595
0.0001559569713071983
0.00013923987965085293
0.00012432704760933488
0.00011102285952965586
9.915283206803345e-05
8.856139174858334e-05
7.91098648574037e-05
7.067466535012738e-05
6.31456673681484e-05
5.6424746008860264e-05
5.042447143154252e-05
4.506694212534692e-05
4.028274355430257e-05
3.601001918083999e-05
3.2193642882531256e-05
2.878448111493858e-05
2.5738738819018375e-05
2.301736976750311e-05
2.0585562762952467e-05
1.841227366505203e-05
1.647009732636953e-05
1.4734148263778101e-05
1.3182214397654561e-05
1.1794654663586968e-05
1.0553942919813837e-05
9.444436170445705e-06
8.452171096119784e-06
7.564681527564076e-06
6.770836691014705e-06
6.0606991281269e-06
5.425387729296574e-06
4.856977427893397e-06
4.348382669160568e-06
3.893276412835248e-06
3.4860031510823107e-06
3.1215109737669223e-06
2.795284109545752e-06
2.503284080753522e-06
2.241904849713046e-06
2.0079207043630637e-06
1.7984473598229776e-06
1.6109043156289632e-06
1.4429883335786674e-06
1.2926350820537814e-06
```

```
1.1580014056712184e-06
1.037436438388734e-06
9.294649648188667e-07
8.3276668236914e-07
7.461586315970762e-07
6.685859440207697e-07
5.991018791164966e-07
5.36860205044418e-07
4.811036780956593e-07
4.311540879734326e-07
3.864052975497157e-07
3.4631272976818724e-07
3.103916419559902e-07
2.7820604666234584e-07
2.493665757525022e-07
2.235242330547715e-07
2.0036662752964725e-07
1.7961406916271733e-07
1.6101610597963257e-07
1.4434841355653347e-07
1.2941010571693734e-07
1.1602128543466011e-07
1.0402082434646952e-07
9.326441852343976e-08
8.362274988135493e-08
7.49799939308504e-08
6.723239527266927e-08
6.028702608399241e-08
5.406062550179954e-08
4.847860158085037e-08
4.347411399612939e-08
3.898727291456688e-08
3.496441211560151e-08
3.135744827221921e-08
2.8123291587559614e-08
2.5223328515894898e-08
2.26229541381511e-08
2.0291155429696614e-08
1.8200137067287912e-08
1.632498637211791e-08
1.464337367650618e-08
1.313528409236892e-08
1.1782776814280018e-08
1.056978340425791e-08
9.481875293010455e-09
8.506129973427988e-09
7.630960687058812e-09
6.845981534564069e-09
6.141882022205125e-09
5.510312745970958e-09
4.943790102561541e-09
4.435605224917734e-09
3.979784496747062e-09
3.570876548574359e-09
3.204046477022082e-09
2.8749568316999748e-09
2.5797181798724397e-09
2.3148435453620055e-09
2.0772039322334064e-09
1.8639959361200393e-09
1.6727033117204262e-09
1.5010698530901528e-09
1.347071344111145e-09
```

```
1.2088951024813536e-09
1.0849106904200139e-09
9.736599966038641e-10
8.738333708142846e-10
7.842541269558106e-10
7.038711045065831e-10
6.317371277104943e-10
5.670057512125597e-10
5.08915453134878e-10
4.5678390119330306e-10
4.0999973929598685e-10
3.6801375426220425e-10
3.303311273300117e-10
2.965134487651615e-10
2.6616089632731117e-10
2.3891982879379247e-10
2.1446961352153604e-10
1.9252572584463677e-10
1.7282859124437483e-10
1.5514900929464294e-10
1.392805601387578e-10
1.250371249717539e-10
1.1225226616102369e-10
1.0077473312033485e-10
9.04737312852764e-11
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:30:
UserWarning: Creating legend with loc="best" can be slow with large amounts of data.
/home/ubuntu/anaconda3/lib/python3.7/site-packages/IPython/core/pylabtools.py:128:
UserWarning: Creating legend with loc="best" can be slow with large amounts of data.
  fig.canvas.print_figure(bytes_io, **kw)
```

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 [23] and that we discuss in a previous lecture.

We discard the first 2000 observations of the simulation and construct the histogram of the part 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 the sample (presumably) from the ergodic where $\mathcal{B}^*$ is the regression coefficient described below; the red vertical line denotes an approximation by [23] to the mean of the ergodic distribution that can be precomputed before sampling from the ergodic distribution, 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.

```
In [9]: titles = ['Government Debt', 'Tax Rate']

        fig, axes = plt.subplots(4, 1, figsize=(10, 15))

        for i, id in enumerate([2, 3]):
            axes[i].plot(sim_seq_long[id][:99], '-k', sim_bel_long[id][:99],
                         '-.b', alpha=0.5)
            axes[i+2].plot(range(100, 199), sim_seq_long[id][100:199], '-k',
                           range(100, 199), sim_bel_long[id][100:199], '-.b',
                           alpha=0.5)
            axes[i].set(title=titles[i])
```

```
    axes[i+2].set(title=titles[i])
    axes[i].grid()
    axes[i+2].grid()

axes[0].legend(('Complete Markets', 'Incomplete Markets'))
plt.tight_layout()
plt.show()
```

Government Debt
Complete Markets
Incomplete Markets
Tax Rate
Government Debt
Tax Rate

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 distributions 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 initial value of the par value of government debt

101.5 Asymptotic Mean and Rate of Convergence

We apply the results of [23] 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 [23].

As in Fiscal Insurance via Fluctuating Interest Rates, we recall that [23] used a particular notation to represent what we can regard as a generalization of the AMSS model.

We introduce some of the [23] notation so that readers can quickly relate notation that appears in their key 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

$$\begin{aligned}
\mathcal{R}_t &= \frac{u_{c,t}}{u_{c,t-1}} R_{t-1} = \frac{u_{c,t}}{\beta E_{t-1} u_{c,t}} \\
B_t &= \frac{b_{t+1}(s^t)}{R_t(s^t)} \\
b_t(s^{t-1}) &= \mathcal{R}_{t-1} B_{t-1} \\
\mathcal{B}_t &= u_{c,t} B_t = (\beta E_t u_{c,t+1}) b_{t+1}(s^t) \\
\mathcal{X}_t &= u_{c,t}[g_t - \tau_t n_t]
\end{aligned}$$

[23] call $\mathcal{X}_t$ the **effective** government deficit, and $\mathcal{B}_t$ the **effective** government debt.

Equation (44) of [23] expresses the time t state s government budget constraint as

$$\mathcal{B}(s) = \mathcal{R}_\tau(s, s_-)\mathcal{B}_- + \mathcal{X}_\tau(s) \tag{1}$$

where the dependence on τ is 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 (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 $\mathcal{X}_t$

101.5.1 Asymptotic Mean

BEGS give conditions under which the ergodic mean of $\mathcal{B}_t$ approximately satisfies the equation

$$\mathcal{B}^* = -\frac{\text{cov}^\infty(\mathcal{R}_\text{t}, \mathcal{X}_\text{t})}{\text{var}^\infty(\mathcal{R}_\text{t})} \tag{2}$$

where the superscript ∞ denotes a moment taken with respect to an ergodic distribution.

Formula (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:

$$\mathcal{B}^* = \text{argmin}_{\mathcal{B}} \text{var}^\infty(\mathcal{R}\mathcal{B} + \mathcal{X}) \tag{3}$$

The minimand in criterion (3) measures fiscal risk associated with a given tax-debt policy that appears on the right side of equation (1).

Expressing formula (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 approximately equal

$$\hat{b} = \frac{\mathcal{B}^*}{\beta E(E_t u_{c,t+1})} = \frac{\mathcal{B}^*}{\beta E(u_{c,t+1})} \tag{4}$$

where mathematical expectations are taken with respect to the ergodic distribution.

101.5.2 Rate of Convergence

BEGS also derive the following approximation to the rate of convergence to $\mathcal{B}^*$ from an arbitrary initial condition.

$$\frac{E_t(\mathcal{B}_{t+1} - \mathcal{B}^*)}{(\mathcal{B}_t - \mathcal{B}^*)} \approx \frac{1}{1 + \beta^2 \text{var}^\infty(\mathcal{R})} \tag{5}$$

(See the equation above equation (47) in [23])

101.5.3 More Advanced Material

The remainder of this lecture is about technical material based on formulas from [23].

The topic is interpreting and extending formula (3) for the ergodic mean $\mathcal{B}^*$.

101.5.4 Chicken and Egg

Attributes of the ergodic distribution for $\mathcal{B}_t$ appear on the right side of formula (3) for the ergodic mean $\mathcal{B}^*$.

Thus, formula (3) is not useful for estimating the mean of the ergodic in advance of actually computing the ergodic distribution

- we need to know the ergodic distribution to compute the right side of formula (3)

So the primary use of equation (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

101.5.5 Approximating the Ergodic Mean

[23] propose an approximation to $\mathcal{B}^*$ that can be computed without first knowing 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 [23]

- 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 [23] text that we'll want to correct

Here is a step-by-step description of the [23] approximation procedure.

101.5.6 Step by Step

Step 1: For a given τ 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 [23] – 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 τ, we want to compute the random variables

$$\mathcal{R}_\tau(s) = \frac{c_\tau(s)^{-\sigma}}{\beta \sum_{s'=1}^{S} c_\tau(s')^{-\sigma} \pi(s')}$$

and

$$\mathcal{X}_\tau(s) = (c_\tau(s) + g(s))^{1+\gamma} - c_\tau(s)^{1-\sigma}$$

each for $s = 1, \dots, S$.

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 τ that satisfies

$$\mathcal{B} = -\frac{\beta}{1-\beta} E\mathcal{X}_\tau \equiv -\frac{\beta}{1-\beta} \sum_s \mathcal{X}_\tau(s)\pi(s)$$

This equation says that at a constant discount factor β, equivalent government debt $\mathcal{B}$ equals the present value of the mean effective government **surplus**.

Typo alert: there is a sign error in equation (46) of [23] –the left side should be multiplied by -1.

- We have made this correction in the above equation.

For a given $\mathcal{B}$, let a τ that solves the above equation be called $\tau(\mathcal{B})$.

We'll use a Python root solver to finds a τ that 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 machine 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 $\text{var}(J(\mathcal{B}))$.

We put $\text{var}(J(\mathcal{B}))$ into a 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 and 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

- this is 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
- so while the approximation circumvents the chicken and egg problem surrounding the much better approximation associated with the green vertical line, it does so by enlarging the approximation error

101.5.7 Execution

Now let's move on to compute things step by step.

Step 1

In [10]:
```
u = CRRAutility(π=(1 / 3) * np.ones((3, 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
```

Out[10]:
```
array([0.93852387, 0.89231015, 0.84858872])
```

In [11]:
```
root(solve_c, np.ones(S), args=(τ, u))
```

Out[11]:
```
     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

```
In [12]: n = c + u.G   # Compute labor supply
```

101.5.8 Note about Code

Remember that in our code π is a 3×3 transition matrix.

But because we are studying an IID case, π has identical rows and we only need to compute objects for one row of π.

This explains why at some places below we set $s = 0$ just to pick off the first row of π in the calculations.

101.5.9 Code

First, let's compute $\mathcal{R}$ and $\mathcal{X}$ according to our formulas

```
In [13]: 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
```

```
In [14]: c**(-u.σ) @ u.π
```

```
Out[14]: array([1.25997521, 1.25997521, 1.25997521])
```

```
In [15]: u.π
```

```
Out[15]: array([[0.33333333, 0.33333333, 0.33333333],
                [0.33333333, 0.33333333, 0.33333333],
                [0.33333333, 0.33333333, 0.33333333]])
```

We only want unconditional expectations because we are in an IID case.

So we'll set $s = 0$ and just pick off expectations associated with the first row of π

```
In [16]: s = 0

         R, X = compute_R_X(τ, u, s)
```

Let's look at the random variables $\mathcal{R}, \mathcal{X}$

```
In [17]: R
```

```
Out[17]: array([1.00116313, 1.10755123, 1.22461897])
```

```
In [18]: mean(R, s)
```

```
Out[18]: 1.1111111111111112
```

```
In [19]: X
```

```
Out[19]: array([0.05457803, 0.18259396, 0.33685546])
```

```
In [20]: mean(X, s)
```

```
Out[20]: 0.19134248445303795
```

```
In [21]: X @ u.π
```

```
Out[21]: array([0.19134248, 0.19134248, 0.19134248])
```

Step 3

```
In [22]: def solve_τ(τ, B, u, s):
             R, X = compute_R_X(τ, u, s)
             return ((u.β - 1) / u.β) * B - X @ u.π[s]
```

Note that B is a scalar.

Let's try out our method computing τ

```
In [23]: s = 0
         B = 1.0

         τ = root(solve_τ, .1, args=(B, u, s)).x[0]  # Very sensitive to initial
 ↪value
         τ
```

```
Out[23]: 0.2740159773695818
```

In the above cell, B is fixed at 1 and τ is to be computed as a function of B.

Note that 0.2 is the initial value for τ in the root-finding algorithm.

Step 4

```
In [24]: 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)
```

```
In [25]: min_J(B, u, s)
```

```
Out[25]: 0.035564405653720765
```

Step 6

In [26]:
```
B_star = minimize(min_J, .5, args=(u, s)).x[0]
B_star
```

Out[26]:
```
-1.199483167941158
```

In [27]:
```
n = c + u.G  # Compute labor supply
```

In [28]:
```
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])
```

In [29]:
```
B_hat = B_star/div
B_hat
```

Out[29]:
```
-1.0577661126390971
```

In [30]:
```
τ_star = root(solve_τ, 0.05, args=(B_star, u, s)).x[0]
τ_star
```

Out[30]:
```
0.09572916798461703
```

In [31]:
```
R_star, X_star = compute_R_X(τ_star, u, s)
R_star, X_star
```

Out[31]:
```
(array([0.9998398 , 1.10746593, 1.2260276 ]),
 array([0.0020272 , 0.12464752, 0.27315299]))
```

In [32]:
```
rate = 1 / (1 + u.β**2 * variance(R_star, s))
rate
```

Out[32]:
```
0.9931353432732218
```

In [33]:
```
root(solve_c, np.ones(S), args=(τ_star, u)).x
```

Out[33]:
```
array([0.9264382 , 0.88027117, 0.83662635])
```

Chapter 102

Competitive Equilibria of a Model of Chang

102.1 Contents

- Overview 102.2
- Setting 102.3
- Competitive Equilibrium 102.4
- Inventory of Objects in Play 102.5
- Analysis 102.6
- Calculating all Promise-Value Pairs in CE 102.7
- Solving a Continuation Ramsey Planner's Bellman Equation 102.8

Co-author: Sebastian Graves

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install polytope
```

102.2 Overview

This lecture describes how Chang [34] 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 [2] to describe key features of the set of competitive equilibria

Roberto Chang [34] chose a model of Calvo [29] as a simple structure that conveys ideas that apply more broadly.

A textbook version of Chang's model appears in chapter 25 of [108].

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 inconsis-

tency, 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:

In [2]:
```
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`
```

102.2.1 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 [29] 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 $t \geq 0$.

102.3 Setting

102.3.1 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

$$\sum_{t=0}^{\infty} \beta^t \left[u(c_t) + v(q_t M_t)\right] \tag{1}$$

subject to

$$q_t M_t \leq y_t + q_t M_{t-1} - c_t - x_t \tag{2}$$

and

$$q_t M_t \leq \bar{m} \tag{3}$$

Here q_t is the reciprocal of the price level at t, which we can also call the *value of money*.

Chang [34] 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 (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 (3) imposes an exogenous upper bound $\bar{m}$ on the household's choice of real balances, where $\bar{m} \geq m^f$.

102.3.2 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

$$-x_t = m_t(1 - h_t) \tag{4}$$

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:

$$y_t = f(x_t), \tag{5}$$

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 (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*.

102.3.3 Household's Problem

Given M_{-1} and $\{q_t\}_{t=0}^{\infty}$, the household's problem is

$$\mathcal{L} = \max_{\vec{c},\vec{M}} \min_{\vec{\lambda},\vec{\mu}} \sum_{t=0}^{\infty} \beta^t \{u(c_t) + v(M_t q_t) + \lambda_t[y_t - c_t - x_t + q_t M_{t-1} - q_t M_t] \\ + \mu_t[\bar{m} - q_t M_t]\}$$

First-order conditions with respect to c_t and M_t, respectively, are

$$u'(c_t) = \lambda_t$$
$$q_t[u'(c_t) - v'(M_t q_t)] \leq \beta u'(c_{t+1}) q_{t+1}, \quad = \text{ if } M_t q_t < \bar{m}$$

The last equation expresses Karush-Kuhn-Tucker complementary slackness conditions (see here).

These insist that the inequality is an equality at an interior solution for M_t.

Using $h_t = \frac{M_{t-1}}{M_t}$ and $q_t = \frac{m_t}{M_t}$ in these first-order conditions and rearranging implies

$$m_t[u'(c_t) - v'(m_t)] \leq \beta u'(f(x_{t+1})) m_{t+1} h_{t+1}, \quad = \text{ if } m_t < \bar{m} \tag{6}$$

Define the following key variable

$$\theta_{t+1} \equiv u'(f(x_{t+1})) m_{t+1} h_{t+1} \tag{7}$$

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 (7) shows that θ_{t+1} intermediates the influences of $(\vec{x}_{t+1}, \vec{m}_{t+1})$ on the household's choice of real balances m_t.

By "intermediates" we mean that the future paths $(\vec{x}_{t+1}, \vec{m}_{t+1})$ influence m_t entirely through their effects on the scalar θ_{t+1}.

The observation that the one dimensional promised marginal utility of real balances θ_{t+1} functions in this way is an important step in constructing a class of competitive equilibria that have a recursive representation.

A closely related observation pervaded the analysis of Stackelberg plans in lecture dynamic Stackelberg problems.

102.4 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.

102.5 Inventory of Objects in Play

Chang constructs the following objects

1. A set Ω of initial marginal utilities of money θ_0

- Let Ω denote the set of initial promised marginal utilities of money θ_0 associated with competitive equilibria.
- Chang exploits the fact that a competitive equilibrium consists of a first period outcome (h_0, m_0, x_0) and a continuation competitive equilibrium with marginal utility of money $\theta_1 \in \Omega$.

1. Competitive equilibria that have a recursive representation

- A competitive equilibrium with a recursive representation consists of an initial θ_0 and a four-tuple of functions (h, m, x, Ψ) mapping θ into this period's (h, m, x) and next period's θ, respectively.
- A competitive equilibrium can be represented recursively by iterating on

$$\begin{aligned} h_t &= h(\theta_t) \\ m_t &= m(\theta_t) \\ x_t &= x(\theta_t) \\ \theta_{t+1} &= \Psi(\theta_t) \end{aligned} \tag{8}$$

  starting from θ_0
  The range and domain of $\Psi(\cdot)$ are both Ω

1. A recursive representation of a Ramsey plan

- A recursive representation of a Ramsey plan is a recursive competitive equilibrium $\theta_0, (h, m, x, \Psi)$ that, among all recursive competitive equilibria, maximizes $\sum_{t=0}^{\infty} \beta^t \left[u(c_t) + v(q_t M_t)\right]$.
- The Ramsey planner chooses $\theta_0, (h, m, x, \Psi)$ from among the set of recursive competitive equilibria at time 0.
- Iterations on the function Ψ determine subsequent θ_t's that summarize the aspects of the continuation competitive equilibria that influence the household's decisions.
- At time 0, the Ramsey planner commits to this implied sequence $\{\theta_t\}_{t=0}^{\infty}$ and therefore to an associated sequence of continuation competitive equilibria.

1. A characterization of time-inconsistency of a Ramsey plan

- Imagine that after a 'revolution' at time $t \geq 1$, a new Ramsey planner is given the opportunity to ignore history and solve a brand new Ramsey plan.

- This new planner would want to reset the θ_t associated with the original Ramsey plan to θ_0.
- The incentive to reinitialize θ_t associated with this revolution experiment indicates the time-inconsistency of the Ramsey plan.
- By resetting θ to θ_0, the new planner avoids the costs at time t that the original Ramsey planner must pay to reap the beneficial effects that the original Ramsey plan for $s \geq t$ had achieved via its influence on the household's decisions for $s = 0, \dots, t-1$.

102.6 Analysis

A competitive equilibrium is a triple of sequences $(\vec{m}, \vec{x}, \vec{h}) \in E^\infty$ that satisfies (2), (3), and (6).

Chang works with a set of competitive equilibria defined as follows.

Definition: $CE = \{(\vec{m}, \vec{x}, \vec{h}) \in E^\infty$ such that (2), (3), and (6) are satisfied $\}$.

CE is not empty because there exists a competitive equilibrium with $h_t = 1$ for all $t \geq 1$, namely, an equilibrium with a constant money supply and constant price level.

Chang establishes that CE is also compact.

Chang makes the following key observation that combines ideas of Abreu, Pearce, and Stacchetti [2] with insights of Kydland and Prescott [102].

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 (2), (3), and (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

$$\Omega = \Big\{ \theta \in \mathbb{R} \ \text{ such that } \ \theta = u'(f(x_0))(m_0 + x_0) \ \text{ for some } \ (\vec{m}, \vec{x}, \vec{h}) \in CE \Big\}$$

Equation (6) inherits from the household's Euler equation for money holdings the property that the value of m_0 consistent with the representative household's choices depends on $(\vec{h}_1, \vec{m}_1)$.

This dependence is captured in the definition above by making Ω be the set of first period values of θ_0 satisfying $\theta_0 = u'(f(x_0))(m_0 + x_0)$ for first period component (m_0, h_0) of competitive equilibrium sequences $(\vec{m}, \vec{x}, \vec{h})$.

Chang establishes that Ω is a nonempty and compact subset of $\mathbb{R}_+$.

Next Chang advances:

Definition: $\Gamma(\theta) = \{(\vec{m}, \vec{x}, \vec{h}) \in CE | \theta = u'(f(x_0))(m_0 + x_0)\}$.

Thus, $\Gamma(\theta)$ is the set of competitive equilibrium sequences $(\vec{m}, \vec{x}, \vec{h})$ whose first period components (m_0, h_0) deliver the prescribed value θ for first period marginal utility.

If we knew the sets $\Omega, \Gamma(\theta)$, we could use the following two-step procedure to find at least the *value* of the Ramsey outcome to the representative household

1. Find the indirect value function $w(\theta)$ defined as

$$w(\theta) = \max_{(\vec{m},\vec{x},\vec{h})\in\Gamma(\theta)} \sum_{t=0}^{\infty} \beta^t \left[u(f(x_t)) + v(m_t)\right]$$

1. 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

$$w(\theta) = \max_{x,m,h,\theta'} \left\{u(f(x)) + v(m) + \beta w(\theta')\right\} \tag{9}$$

where maximization is subject to

$$(m, x, h) \in E \text{ and } \theta' \in \Omega \tag{10}$$

and

$$\theta = u'(f(x))(m + x) \tag{11}$$

and

$$-x = m(1 - h) \tag{12}$$

and

$$m \cdot [u'(f(x)) - v'(m)] \leq \beta\theta', \quad = \text{ if } m < \bar{m} \tag{13}$$

Before we use this proposition to recover a recursive representation of the Ramsey plan, note that the proposition relies on knowing the set Ω.

To find Ω, Chang uses the insights of Kydland and Prescott [102] together with a method based on the Abreu, Pearce, and Stacchetti [2] iteration to convergence on an operator B that maps continuation values into values.

We want an operator that maps a continuation θ into a current θ.

Chang lets Q be a nonempty, bounded subset of $\mathbb{R}$.

Elements of the set Q are taken to be candidate values for continuation marginal utilities.

Chang defines an operator

$$B(Q) = \theta \in \mathbb{R} \text{ such that there is } (m, x, h, \theta') \in E \times Q$$

such that (11), (12), and (13) hold.

Thus, $B(Q)$ is the set of first period θ's attainable with $(m, x, h) \in E$ and some $\theta' \in Q$.

Proposition:

1. $Q \subset B(Q)$ implies $B(Q) \subset \Omega$ ('self-generation').
2. $\Omega = B(\Omega)$ ('factorization').

The proposition characterizes Ω as the largest fixed point of B.

It is easy to establish that $B(Q)$ is a monotone operator.

This property allows Chang to compute Ω as the limit of iterations on B provided that iterations begin from a sufficiently large initial set.

102.6.1 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_π is the set of money growth sequences consistent with the existence of competitive equilibria.

Chang observes that CE_π is nonempty and compact.

Definition: σ is said to be *admissible* if for all $t \geq 1$ and after any history $\vec{h}^{t-1}$, the continuation $\vec{h}_t$ implied by σ belongs to CE_π.

Admissibility of σ means that anticipated policy choices associated with σ are consistent with the existence of competitive equilibria after each possible subsequent history.

After any history $\vec{h}^{t-1}$, admissibility restricts the government's choice in period t to the set

$$CE_\pi^0 = \{h \in \Pi : \text{there is } \vec{h} \in CE_\pi \text{ with } h = h_0\}$$

In words, CE_π^0 is the set of all first period money growth rates $h = h_0$, each of which is consistent with the existence of a sequence of money growth rates $\vec{h}$ starting from h_0 in the initial period and for which a competitive equilibrium exists.

Remark: $CE_\pi^0 = \{h \in \Pi : \text{ there is } (m, \theta') \in [0, \bar{m}] \times \Omega \text{ such that } mu'[f((h-1)m) - v'(m)] \leq \beta\theta' \text{ with equality if } m < \bar{m}\}$.

Definition: An *allocation rule* is a sequence of functions $\vec{\alpha} = \{\alpha_t\}_{t=0}^{\infty}$ such that $\alpha_t : \Pi^t \to [0, \bar{m}] \times X$.

Thus, the time t component of $\alpha_t(h^t)$ is a pair of functions $(m_t(h^t), x_t(h^t))$.

Definition: Given an admissible government strategy σ, an allocation rule α is called *competitive* if given any history $\vec{h}^{t-1}$ and $h_t \in CE_\pi^0$, the continuations of σ and α after $(\vec{h}^{t-1}, h_t)$ induce a competitive equilibrium sequence.

102.6.2 Another Operator

At this point it is convenient to introduce another operator that can be used to compute a Ramsey plan.

For computing a Ramsey plan, this operator is wasteful because it works with a state vector that is bigger than necessary.

We introduce this operator because it helps to prepare the way for Chang's operator called $\tilde{D}(Z)$ that we shall describe in lecture credible government policies.

It is also useful because a fixed point of the operator to be defined here provides a good guess for an initial set from which to initiate iterations on Chang's set-to-set operator $\tilde{D}(Z)$ to be described in lecture credible government policies.

Let S be the set of all pairs (w, θ) of competitive equilibrium values and associated initial marginal utilities.

Let W be a bounded set of *values* in $\mathbb{R}$.

Let Z be a nonempty subset of $W \times \Omega$.

Think of using pairs (w', θ') drawn from Z as candidate continuation value, θ pairs.

Define the operator

$$D(Z) = \Big\{(w, \theta) : \text{there is } h \in CE_\pi^0$$

$$\text{and a four-tuple } \ (m(h), x(h), w'(h), \theta'(h)) \in [0, \bar{m}] \times X \times Z$$

such that

$$w = u(f(x(h))) + v(m(h)) + \beta w'(h) \tag{14}$$

$$\theta = u'(f(x(h)))(m(h) + x(h)) \tag{15}$$

$$x(h) = m(h)(h - 1) \tag{16}$$

$$m(h)(u'(f(x(h))) - v'(m(h))) \leq \beta\theta'(h) \tag{17}$$

$$\text{with equality if } m(h) < \bar{m}\Big\}$$

It is possible to establish.

Proposition:

1. If $Z \subset D(Z)$, then $D(Z) \subset S$ ('self-generation').
2. $S = D(S)$ ('factorization').

Proposition:

1. Monotonicity of D: $Z \subset Z'$ implies $D(Z) \subset D(Z')$.
2. Z compact implies that $D(Z)$ is compact.

It can be shown that S is compact and that therefore there exists a (w, θ) pair within this set that attains the highest possible value w.

This (w, θ) pair i associated with a Ramsey plan.

Further, we can compute S by iterating to convergence on D provided that one begins with a sufficiently large initial set S_0.

As a very useful by-product, the algorithm that finds the largest fixed point $S = D(S)$ also produces the Ramsey plan, its value w, and the associated competitive equilibrium.

102.7 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 [94].

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, $C_{i,t+1}$.

1. 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

$$H \cdot (w', \theta') \leq C_t$$

$$w = u(f(x_j)) + v(m_j) + \beta w'$$

$$\theta = u'(f(x_j))(m_j + x_j)$$

$$x_j = m_j(h_j - 1)$$

$$m_j(u'(f(x_j)) - v'(m_j)) \leq \beta\theta' \;\; (= \text{if } m_j < \bar{m})$$

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)

In [3]:
```
"""
Author: Sebastian Graves

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:
    """
    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, ϑ) space for initial guess of equilibrium sets
        p_vec = np.zeros(self.N_a)
        w_vec = np.zeros(self.N_a)
        for i in range(self.N_a):
            p_vec[i] = self.Θ_vec[i]
            w_vec[i] = self.u_vec[i]/(1 - β)

        w_space = np.array([min(w_vec[~np.isinf(w_vec)]),
                            max(w_vec[~np.isinf(w_vec)])])
        p_space = np.array([0, max(p_vec[~np.isinf(w_vec)])])
        self.p_space = p_space

        # Set up hyperplane levels and gradients for iterations
```

```
        def SG_H_V(N, w_space, p_space):
            """
            This function  initializes the subgradients, hyperplane levels,
            and extreme points of the value set by choosing an appropriate
            origin and radius. It is based on a similar function in
↪QuantEcon's
            Games.jl
            """

            # First, create a unit circle. Want points placed on [0, 2π]
            inc = 2 * np.pi / N
            degrees = np.arange(0, 2 * np.pi, inc)

            # Points on circle
            H = np.zeros((N, 2))
            for i in range(N):
                x = degrees[i]
                H[i, 0] = np.cos(x)
                H[i, 1] = np.sin(x)

            # Then calculate origin and radius
            o = np.array([np.mean(w_space), np.mean(p_space)])
            r1 = max((max(w_space) - o[0])**2, (o[0] - min(w_space))**2)
            r2 = max((max(p_space) - o[1])**2, (o[1] - min(p_space))**2)
            r = np.sqrt(r1 + r2)

            # Now calculate vertices
            Z = np.zeros((2, N))
            for i in range(N):
                Z[0, i] = o[0] + r*H.T[0, i]
                Z[1, i] = o[1] + r*H.T[1, i]

            # Corresponding hyperplane levels
            C = np.zeros(N)
            for i in range(N):
                C[i] = np.dot(Z[:, i], H[i, :])

            return C, H, Z

        C, self.H, Z = SG_H_V(N_g, w_space, p_space)
        C = C.reshape(N_g, 1)
        self.c0_c, self.c0_s, self.c1_c, self.c1_s = np.copy(C), np.
↪copy(C), \
            np.copy(C), np.copy(C)
        self.z0_s, self.z0_c, self.z1_s, self.z1_c = np.copy(Z), np.
↪copy(Z), \
            np.copy(Z), np.copy(Z)

        self.w_bnds_s, self.w_bnds_c = (w_space[0], w_space[1]), \
            (w_space[0], w_space[1])
        self.p_bnds_s, self.p_bnds_c = (p_space[0], p_space[1]), \
            (p_space[0], p_space[1])

        # Create dictionaries to save equilibrium set for each iteration
        self.c_dic_s, self.c_dic_c = {}, {}
        self.c_dic_s[0], self.c_dic_c[0] = self.c0_s, self.c0_c

    def solve_worst_spe(self):
```

```
        """
        Method to solve for BR(Z). See p.449 of Chang (1998)
        """

        p_vec = np.full(self.N_a, np.nan)
        c = [1, 0]

        # Pre-compute constraints
        aineq_mbar = np.vstack((self.H, np.array([0, -self.β])))
        bineq_mbar = np.vstack((self.c0_s, 0))

        aineq = self.H
        bineq = self.c0_s
        aeq = [[0, -self.β]]

        for j in range(self.N_a):
            # Only try if consumption is possible
            if self.f_vec[j] > 0:
                # If m = mbar, use inequality constraint
                if self.A[j, 1] == self.mbar:
                    bineq_mbar[-1] = self.euler_vec[j]
                    res = linprog(c, A_ub=aineq_mbar, b_ub=bineq_mbar,
                                  bounds=(self.w_bnds_s, self.p_bnds_s))
                else:
                    beq = self.euler_vec[j]
                    res = linprog(c, A_ub=aineq, b_ub=bineq, A_eq=aeq,
↪b_eq=beq,
                                  bounds=(self.w_bnds_s, self.p_bnds_s))
                if res.status == 0:
                    p_vec[j] = self.u_vec[j] + self.β * res.x[0]

        # Max over h and min over other variables (see Chang (1998) p.449)
        self.br_z = np.nanmax(np.nanmin(p_vec.reshape(self.n_m, self.n_h),
↪0))

    def solve_subgradient(self):
        """
        Method to solve for E(Z). See p.449 of Chang (1998)
        """

        # Pre-compute constraints
        aineq_C_mbar = np.vstack((self.H, np.array([0, -self.β])))
        bineq_C_mbar = np.vstack((self.c0_c, 0))

        aineq_C = self.H
        bineq_C = self.c0_c
        aeq_C = [[0, -self.β]]

        aineq_S_mbar = np.vstack((np.vstack((self.H, np.array([0, -self.
↪β]))),
                                  np.array([-self.β, 0])))
        bineq_S_mbar = np.vstack((self.c0_s, np.zeros((2, 1))))

        aineq_S = np.vstack((self.H, np.array([-self.β, 0])))
        bineq_S = np.vstack((self.c0_s, 0))
        aeq_S = [[0, -self.β]]

        # Update maximal hyperplane level
```

```
        for i in range(self.N_g):
            c_a1a2_c, t_a1a2_c = np.full(self.N_a, -np.inf), \
                np.zeros((self.N_a, 2))
            c_a1a2_s, t_a1a2_s = np.full(self.N_a, -np.inf), \
                np.zeros((self.N_a, 2))

            c = [-self.H[i, 0], -self.H[i, 1]]

            for j in range(self.N_a):
                # Only try if consumption is possible
                if self.f_vec[j] > 0:

                    # COMPETITIVE EQUILIBRIA
                    # If m = mbar, use inequality constraint
                    if self.A[j, 1] == self.mbar:
                        bineq_C_mbar[-1] = self.euler_vec[j]
                     res = linprog(c, A_ub=aineq_C_mbar, b_ub=bineq_C_mbar,
                                    bounds=(self.w_bnds_c, self.p_bnds_c))
                    # If m < mbar, use equality constraint
                    else:
                        beq_C = self.euler_vec[j]
                       res = linprog(c, A_ub=aineq_C, b_ub=bineq_C, A_eq =
↪aeq_C,
                                    b_eq = beq_C, bounds=(self.w_bnds_c, \
                                         self.p_bnds_c))
                    if res.status == 0:
                        c_a1a2_c[j] = self.H[i, 0] * (self.u_vec[j] \
                         + self.β * res.x[0]) + self.H[i, 1] * self.Θ_vec[j]
                        t_a1a2_c[j] = res.x

                    # SUSTAINABLE EQUILIBRIA
                    # If m = mbar, use inequality constraint
                    if self.A[j, 1] == self.mbar:
                        bineq_S_mbar[-2] = self.euler_vec[j]
                        bineq_S_mbar[-1] = self.u_vec[j] - self.br_z
                     res = linprog(c, A_ub=aineq_S_mbar, b_ub=bineq_S_mbar,
                                    bounds=(self.w_bnds_s, self.p_bnds_s))
                    # If m < mbar, use equality constraint
                    else:
                        bineq_S[-1] = self.u_vec[j] - self.br_z
                        beq_S = self.euler_vec[j]
                       res = linprog(c, A_ub=aineq_S, b_ub=bineq_S, A_eq =
↪aeq_S,
                                    b_eq = beq_S, bounds=(self.w_bnds_s, \
                                         self.p_bnds_s))
                    if res.status == 0:
                        c_a1a2_s[j] = self.H[i, 0] * (self.u_vec[j] \
                          + self.β*res.x[0]) + self.H[i, 1] * self.Θ_vec[j]
                        t_a1a2_s[j] = res.x

            idx_c = np.where(c_a1a2_c == max(c_a1a2_c))[0][0]
            self.z1_c[:, i] = np.array([self.u_vec[idx_c]
                                        + self.β * t_a1a2_c[idx_c, 0],
                                        self.Θ_vec[idx_c]])

            idx_s = np.where(c_a1a2_s == max(c_a1a2_s))[0][0]
            self.z1_s[:, i] = np.array([self.u_vec[idx_s]
                                        + self.β * t_a1a2_s[idx_s, 0],
```

```
                                        self.Θ_vec[idx_s]])

        for i in range(self.N_g):
            self.c1_c[i] = np.dot(self.z1_c[:, i], self.H[i, :])
            self.c1_s[i] = np.dot(self.z1_s[:, i], self.H[i, :])

    def solve_sustainable(self, tol=1e-5, max_iter=250):
        """
      Method to solve for the competitive and sustainable equilibrium sets.
        """

        t = time.time()
        diff = tol + 1
        iters = 0

        print('### --------------- ###')
        print('Solving Chang Model Using Outer Hyperplane Approximation')
        print('### --------------- ### \n')

        print('Maximum difference when updating hyperplane levels:')

        while diff > tol and iters < max_iter:
            iters = iters + 1
            self.solve_worst_spe()
            self.solve_subgradient()
            diff = max(np.maximum(abs(self.c0_c - self.c1_c),
                       abs(self.c0_s - self.c1_s)))
            print(diff)

            # Update hyperplane levels
            self.c0_c, self.c0_s = np.copy(self.c1_c), np.copy(self.c1_s)

            # Update bounds for w and ϑ
            wmin_c, wmax_c = np.min(self.z1_c, axis=1)[0], \
                np.max(self.z1_c, axis=1)[0]
            pmin_c, pmax_c = np.min(self.z1_c, axis=1)[1], \
                np.max(self.z1_c, axis=1)[1]

            wmin_s, wmax_s = np.min(self.z1_s, axis=1)[0], \
                np.max(self.z1_s, axis=1)[0]
            pmin_S, pmax_S = np.min(self.z1_s, axis=1)[1], \
                np.max(self.z1_s, axis=1)[1]

          self.w_bnds_s, self.w_bnds_c = (wmin_s, wmax_s), (wmin_c, wmax_c)
          self.p_bnds_s, self.p_bnds_c = (pmin_S, pmax_S), (pmin_c, pmax_c)

            # Save iteration
           self.c_dic_c[iters], self.c_dic_s[iters] = np.copy(self.c1_c), \
                np.copy(self.c1_s)
            self.iters = iters

        elapsed = time.time() - t
        print('Convergence achieved after {} iterations and {} \
            seconds'.format(iters, round(elapsed, 2)))

    def solve_bellman(self, θ_min, θ_max, order, disp=False, tol=1e-7,
↪maxiters=100):
        """
```

```
        Continuous Method to solve the Bellman equation in section 25.3
        """
        mbar = self.mbar

        # Utility and production functions
        uc = lambda c: np.log(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

      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))

        # 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

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)
```

In [4]:
```
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]
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-4-d19a06b35f4c> in <module>
      1 ch1 = ChangModel(β=0.3, mbar=30, h_min=0.9, h_max=2, n_h=8, n_m=35, N_g=10)
----> 2 ch1.solve_sustainable()


<ipython-input-3-ae015887b922> in solve_sustainable(self, tol, max_iter)
```

```
      271             iters = iters + 1
      272             self.solve_worst_spe()
  --> 273             self.solve_subgradient()
      274             diff = max(np.maximum(abs(self.c0_c - self.
↪c1_c),
      275                        abs(self.c0_s - self.c1_s)))


      <ipython-input-3-ae015887b922> in solve_subgradient(self)
      233                         res = linprog(c, A_ub=aineq_S,
↪b_ub=bineq_S, A_eq = aeq_S,
      234                                       b_eq = beq_S,
↪bounds=(self.w_bnds_s, \
  --> 235                                           self.p_bnds_s))
      236                     if res.status == 0:
      237                       c_a1a2_s[j] = self.H[i, 0] * (self.
↪u_vec[j] \


      ~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_linprog.py in linprog(c, A_ub, b_ub, A_eq, b_eq, bounds, method,
↪callback, options, x0)
      539     x, fun, slack, con, status, message = _postprocess(
      540         x, c_o, A_ub_o, b_ub_o, A_eq_o, b_eq_o, bounds,
  --> 541         complete, undo, status, message, tol, iteration,
↪disp)
      542
      543     sol = {


      ~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_linprog_util.py in _postprocess(x, c, A_ub, b_ub, A_eq, b_eq,
↪bounds, complete, undo, status, message, tol, iteration, disp)
     1412     status, message = _check_result(
     1413         x, fun, status, slack, con,
  -> 1414         lb, ub, tol, message
     1415     )
     1416


      ~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_linprog_util.py in _check_result(x, fun, status, slack, con, lb,
↪ub, tol, message)
     1325         # nearly basic feasible solution. Postsolving can
↪make the solution
     1326         # basic, however, this solution is NOT optimal
  -> 1327         raise ValueError(message)
     1328
     1329     return status, message
```

```
ValueError: The algorithm terminated successfully and
↪determined that the problem is infeasible.
```

```
In [5]: 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
            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)
```

```
In [6]: 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]
```

```
↪----------------------------------------------------------------------------------------
```

```
ValueError                                Traceback (most
↪recent call last)

<ipython-input-6-1970f7c91f36> in <module>
  1 ch2 = ChangModel(β=0.8, mbar=30, h_min=0.9, h_max=1/0.8,
  2                  n_h=8, n_m=35, N_g=10)
----> 3 ch2.solve_sustainable()


<ipython-input-3-ae015887b922> in solve_sustainable(self,
↪tol, max_iter)
    271             iters = iters + 1
    272             self.solve_worst_spe()
--> 273             self.solve_subgradient()
    274             diff = max(np.maximum(abs(self.c0_c - self.
↪c1_c),
    275                       abs(self.c0_s - self.c1_s)))


<ipython-input-3-ae015887b922> in solve_subgradient(self)
    233                        res = linprog(c, A_ub=aineq_S,
↪b_ub=bineq_S, A_eq = aeq_S,
    234                                      b_eq = beq_S,
↪bounds=(self.w_bnds_s, \
--> 235                                          self.p_bnds_s))
    236                     if res.status == 0:
    237                       c_a1a2_s[j] = self.H[i, 0] * (self.
↪u_vec[j] \


~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_linprog.py in linprog(c, A_ub, b_ub, A_eq, b_eq, bounds, method,
↪callback, options, x0)
    539     x, fun, slack, con, status, message = _postprocess(
    540         x, c_o, A_ub_o, b_ub_o, A_eq_o, b_eq_o, bounds,
--> 541         complete, undo, status, message, tol, iteration,
↪disp)
    542
    543     sol = {


~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_linprog_util.py in _postprocess(x, c, A_ub, b_ub, A_eq, b_eq,
↪bounds, complete, undo, status, message, tol, iteration, disp)
   1412     status, message = _check_result(
   1413         x, fun, status, slack, con,
-> 1414         lb, ub, tol, message
   1415     )
   1416
```

```
~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_linprog_util.py in _check_result(x, fun, status, slack, con, lb,
↪ub, tol, message)
     1325         # nearly basic feasible solution. Postsolving can
↪make the solution
     1326         # basic, however, this solution is NOT optimal
   -> 1327         raise ValueError(message)
     1328
     1329     return status, message


ValueError: The algorithm terminated successfully and
↪determined that the problem is infeasible.
```

```
In [7]: plot_competitive(ch2)
```

102.8 Solving a Continuation Ramsey Planner's Bellman Equation

In this section we solve the Bellman equation confronting a **continuation Ramsey planner**.

The construction of a Ramsey plan is decomposed into a two subproblems in Ramsey plans, time inconsistency, sustainable plans and dynamic Stackelberg problems.

- Subproblem 1 is faced by a sequence of continuation Ramsey planners at $t \geq 1$.
- Subproblem 2 is faced by a Ramsey planner at $t = 0$.

The problem is:

$$J(\theta) = \max_{m,x,h,\theta'} u(f(x)) + v(m) + \beta J(\theta')$$

subject to:

$$\theta \leq u'(f(x))x + v'(m)m + \beta\theta'$$

$$\theta = u'(f(x))(m + x)$$

$$x = m(h - 1)$$

$$(m, x, h) \in E$$

$$\theta' \in \Omega$$

To solve this Bellman equation, we must know the set Ω.

We have solved the Bellman equation for the two sets of parameter values for which we computed the equilibrium value sets above.

Hence for these parameter configurations, we know the bounds of Ω.

The two sets of parameters differ only in the level of β.

From the figures earlier in this lecture, we know that when $\beta = 0.3$, $\Omega = [0.0088, 0.0499]$, and when $\beta = 0.8$, $\Omega = [0.0395, 0.2193]$

In [8]:
```
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)
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:35:
RuntimeWarning: invalid value encountered in log
```

In [9]:
```
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)
```

```
    /home/ubuntu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:311:
RuntimeWarning: invalid value encountered in log
```

```
    Convergence achieved after 15 iterations
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:

```
In [10]: max(abs(ch1.resid_grid)), max(abs(ch2.resid_grid))
```

```
Out[10]: (6.463130481471069e-06, 6.875604405820468e-07)
```

The value functions plotted below trace out the right edges of the sets of equilibrium values plotted above

```
In [11]: 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"$\beta = {model.β}$")

         plt.show()
```

The next figure plots the optimal policy functions; values of θ', m, x, h for each value of the state θ:

```
In [12]: for model in (ch1, ch2):

             fig, axes = plt.subplots(2, 2, figsize=(12, 6), sharex=True)
             fig.suptitle(rf"$\beta = {model.β}$", fontsize=16)

             plots = [model.θ_prime_grid, model.m_grid,
                      model.h_grid, model.x_grid]
             labels = [r"$\theta'$", "$m$", "$h$", "$x$"]

             for ax, plot, label in zip(axes.flatten(), plots, labels):
                 ax.plot(model.θ_grid_fine, plot)
                 ax.set_xlabel(r"$\theta$", fontsize=14)
                 ax.set_ylabel(label, fontsize=14)

             plt.show()
```

With the first set of parameter values, the value of θ' chosen by the Ramsey planner quickly hits the upper limit of Ω.

But with the second set of parameters it converges to a value in the interior of the set.

Consequently, the choice of $\bar{\theta}$ is clearly important with the first set of parameter values.

One way of seeing this is plotting $\theta'(\theta)$ for each set of parameters.

With the first set of parameter values, this function does not intersect the 45-degree line until $\bar{\theta}$, whereas in the second set of parameter values, it intersects in the interior.

```
In [13]: fig, axes = plt.subplots(1, 2, figsize=(12, 4))
```

```
    for ax, model in zip(axes, (ch1, ch2)):
        ax.plot(model.θ_grid_fine, model.θ_prime_grid,
↪label=r"$\theta'(\theta)$")
        ax.plot(model.θ_grid_fine, model.θ_grid_fine, label=r"$\theta$")
        ax.set(xlabel=r"$\theta$", title=rf"$\beta = {model.β}$")

    axes[0].legend()
    plt.show()
```

Subproblem 2 is equivalent to the planner choosing the initial value of θ (i.e. the value which maximizes the value function).

From this starting point, we can then trace out the paths for $\{\theta_t, m_t, h_t, x_t\}_{t=0}^{\infty}$ that support this equilibrium.

These are shown below for both sets of parameters

```
In [14]: 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()
```

102.8.1 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 103

Credible Government Policies in a Model of Chang

103.1 Contents

- Overview 103.2
- The Setting 103.3
- Calculating the Set of Sustainable Promise-Value Pairs 103.4

Co-author: Sebastian Graves

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install polytope
```

103.2 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 [35], also referred to as a *credible public policy* in [150].

Technically, this lecture starts where lecture competitive equilibria in the Chang model on

Ramsey plans within the Chang [34] model stopped.

That lecture presents recursive representations of *competitive equilibria* and a *Ramsey plan* for a version of a model of Calvo [29] 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:

In [2]:
```
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`
```

103.3 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.

103.3.1 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

$$\sum_{t=0}^{\infty} \beta^t \left[u(c_t) + v(q_t M_t)\right] \tag{1}$$

subject to

$$q_t M_t \leq y_t + q_t M_{t-1} - c_t - x_t \tag{2}$$

and

$$q_t M_t \leq \bar{m} \tag{3}$$

Here q_t is the reciprocal of the price level at t, also known as the *value of money.*

Chang [34] 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 (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 (3) imposes an exogenous upper bound $\bar{m}$ on the choice of real balances, where $\bar{m} \geq m^f$.

103.3.2 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

$$-x_t = m_t(1 - h_t) \tag{4}$$

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:

$$y_t = f(x_t) \tag{5}$$

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 (1).

103.3.3 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.

103.3.4 Household's Problem

Given M_{-1} and $\{q_t\}_{t=0}^{\infty}$, the household's problem is

$$\mathcal{L} = \max_{\vec{c},\vec{M}} \min_{\vec{\lambda},\vec{\mu}} \sum_{t=0}^{\infty} \beta^t \{u(c_t) + v(M_t q_t) + \lambda_t[y_t - c_t - x_t + q_t M_{t-1} - q_t M_t] + \mu_t[\bar{m} - q_t M_t]\}$$

First-order conditions with respect to c_t and M_t, respectively, are

$$u'(c_t) = \lambda_t$$
$$q_t[u'(c_t) - v'(M_t q_t)] \leq \beta u'(c_{t+1}) q_{t+1}, \quad = \text{ if } M_t q_t < \bar{m}$$

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

$$m_t[u'(c_t) - v'(m_t)] \leq \beta u'(f(x_{t+1})) m_{t+1} h_{t+1}, \quad = \text{ if } m_t < \bar{m} \tag{6}$$

Define the following key variable

$$\theta_{t+1} \equiv u'(f(x_{t+1})) m_{t+1} h_{t+1} \tag{7}$$

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 (7) shows that θ_{t+1} intermediates the influences of $(\vec{x}_{t+1}, \vec{m}_{t+1})$ on the household's choice of real balances m_t.

By "intermediates" we mean that the future paths $(\vec{x}_{t+1}, \vec{m}_{t+1})$ influence m_t entirely through their effects on the scalar θ_{t+1}.

The observation that the one dimensional promised marginal utility of real balances θ_{t+1} functions in this way is an important step in constructing a class of competitive equilibria that have a recursive representation.

A closely related observation pervaded the analysis of Stackelberg plans in dynamic Stackelberg problems and the Calvo model.

103.3.5 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.

103.3.6 A Credible Government Policy

Chang works with

A credible government policy with a recursive representation

- Here there is no time 0 Ramsey planner.
- Instead there is a sequence of governments, one for each t, that choose time t government actions after forecasting what future governments will do.
- Let $w = \sum_{t=0}^{\infty} \beta^t \left[u(c_t) + v(q_t M_t)\right]$ be a value associated with a particular competitive equilibrium.
- A recursive representation of a credible government policy is a pair of initial conditions (w_0, θ_0) and a five-tuple of functions

$$h(w_t, \theta_t), m(h_t, w_t, \theta_t), x(h_t, w_t, \theta_t), \chi(h_t, w_t, \theta_t), \Psi(h_t, w_t, \theta_t)$$

  mapping w_t, θ_t and in some cases h_t into $\hat{h}_t, m_t, x_t, w_{t+1}$, and θ_{t+1}, respectively.
- Starting from an initial condition (w_0, θ_0), a credible government policy can be constructed by iterating on these functions in the following order that respects the within-period timing:

$$\begin{aligned} \hat{h}_t &= h(w_t, \theta_t) \\ m_t &= m(h_t, w_t, \theta_t) \\ x_t &= x(h_t, w_t, \theta_t) \\ w_{t+1} &= \chi(h_t, w_t, \theta_t) \\ \theta_{t+1} &= \Psi(h_t, w_t, \theta_t) \end{aligned} \tag{8}$$

- Here it is to be understood that $\hat{h}_t$ is the action that the government policy instructs the government to take, while h_t possibly not equal to $\hat{h}_t$ is some other action that the government is free to take at time t.

The plan is *credible* if it is in the time t government's interest to execute it.

Credibility requires that the plan be such that for all possible choices of h_t that are consistent with competitive equilibria,

$$\begin{aligned} & u(f(x(\hat{h}_t, w_t, \theta_t))) + v(m(\hat{h}_t, w_t, \theta_t)) + \beta\chi(\hat{h}_t, w_t, \theta_t) \\ & \geq u(f(x(h_t, w_t, \theta_t))) + v(m(h_t, w_t, \theta_t)) + \beta\chi(h_t, w_t, \theta_t) \end{aligned}$$

so that at each instance and circumstance of choice, a government attains a weakly higher lifetime utility with continuation value $w_{t+1} = \Psi(h_t, w_t, \theta_t)$ by adhering to the plan and confirming the associated time t action $\hat{h}_t$ that the public had expected earlier.

Please note the subtle change in arguments of the functions used to represent a competitive equilibrium and a Ramsey plan, on the one hand, and a credible government plan, on the other hand.

The extra arguments appearing in the functions used to represent a credible plan come from allowing the government to contemplate disappointing the private sector's expectation about its time t choice $\hat{h}_t$.

A credible plan induces the government to confirm the private sector's expectation.

The recursive representation of the plan uses the evolution of continuation values to deter the government from wanting to disappoint the private sector's expectations.

Technically, a Ramsey plan and a credible plan both incorporate history dependence.

For a Ramsey plan, this is encoded in the dynamics of the state variable θ_t, a promised marginal utility that the Ramsey plan delivers to the private sector.

For a credible government plan, we the two-dimensional state vector (w_t, θ_t) encodes history dependence.

103.3.7 Sustainable Plans

A government strategy σ and an allocation rule α are said to constitute a *sustainable plan* (SP) if.

1. σ is admissible.
2. Given σ, α is competitive.
3. After any history $\vec{h}^{t-1}$, the continuation of σ is optimal for the government; i.e., the sequence $\vec{h}_t$ induced by σ after $\vec{h}^{t-1}$ maximizes over CE_π given α.

Given any history $\vec{h}^{t-1}$, the continuation of a sustainable plan is a sustainable plan.

Let $\Theta = \{(\vec{m}, \vec{x}, \vec{h}) \in CE : \text{there is an SP whose outcome is}(\vec{m}, \vec{x}, \vec{h})\}$.

Sustainable outcomes are elements of Θ.

Now consider the space

$$S = \Big\{(w, \theta) : \text{there is a sustainable outcome } (\vec{m}, \vec{x}, \vec{h}) \in \Theta$$

with value

$$w = \sum_{t=0}^{\infty} \beta^t [u(f(x_t)) + v(m_t)] \text{ and such that } u'(f(x_0))(m_0 + x_0) = \theta\Big\}$$

The space S is a compact subset of $W \times \Omega$ where $W = [\underline{w}, \overline{w}]$ is the space of values associated with sustainable plans. Here $\underline{w}$ and $\overline{w}$ are finite bounds on the set of values.

Because there is at least one sustainable plan, S is nonempty.

Now recall the within-period timing protocol, which we can depict $(h, x) \to m = qM \to y = c$.

With this timing protocol in mind, the time 0 component of an SP has the following components:

1. A period 0 action $\hat{h} \in \Pi$ that the public expects the government to take, together with subsequent within-period consequences $m(\hat{h}), x(\hat{h})$ when the government acts as expected.
2. For any first-period action $h \neq \hat{h}$ with $h \in CE_\pi^0$, 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 θ being $\theta'(h)$.

Given the timing protocol within the model, the representative household's response to a government deviation to $h \neq \hat{h}$ from a prescribed $\hat{h}$ consists of a first-period action $m(h)$ and associated subsequent actions, together with future equilibrium prices, captured by $(w'(h), \theta'(h))$.

At this point, Chang introduces an idea in the spirit of Abreu, Pearce, and Stacchetti [2].

Let Z be a nonempty subset of $W \times \Omega$.

Think of using pairs (w', θ') drawn from Z as candidate continuation value, promised marginal utility pairs.

Define the following operator:

$$\begin{aligned} \tilde{D}(Z) = \Big\{ (w, \theta) : & \text{ there is } \hat{h} \in CE^0_\pi \text{ and for each } h \in CE^0_\pi \\ & \text{a four-tuple } (m(h), x(h), w'(h), \theta'(h)) \in [0, \bar{m}] \times X \times Z \end{aligned} \tag{9}$$

such that

$$w = u(f(x(\hat{h}))) + v(m(\hat{h})) + \beta w'(\hat{h}) \tag{10}$$

$$\theta = u'(f(x(\hat{h})))(m(\hat{h}) + x(\hat{h})) \tag{11}$$

and for all $h \in CE^0_\pi$

$$w \geq u(f(x(h))) + v(m(h)) + \beta w'(h) \tag{12}$$

$$x(h) = m(h)(h - 1) \tag{13}$$

and

$$m(h)(u'(f(x(h))) - v'(m(h))) \leq \beta \theta'(h) \tag{14}$$

$$\text{with equality if } m(h) < \bar{m} \Big\}$$

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 (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 (8), which we repeat here for convenience:

$$\begin{aligned}
\hat{h}_t &= h(w_t, \theta_t) \\
m_t &= m(h_t, w_t, \theta_t) \\
x_t &= x(h_t, w_t, \theta_t) \\
w_{t+1} &= \chi(h_t, w_t, \theta_t) \\
\theta_{t+1} &= \Psi(h_t, w_t, \theta_t)
\end{aligned}$$

103.4 Calculating the Set of Sustainable Promise-Value Pairs

Above we defined the $\tilde{D}(Z)$ operator as (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, $C_{i,t+1}$.

1. 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

$$H \cdot (w', \theta') \leq C_t$$

$$x_j = m_j(h_j - 1)$$

$$m_j(u'(f(x_j)) - v'(m_j)) \leq \beta\theta' \ \ (= \text{if } m_j < \bar{m})$$

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

$$H \cdot (w', \theta') \leq C_t$$

$$w = u(f(x_j)) + v(m_j) + \beta w'$$

$$\theta = u'(f(x_j))(m_j + x_j)$$

$$x_j = m_j(h_j - 1)$$

$$m_j(u'(f(x_j)) - v'(m_j)) \leq \beta\theta' \ \ (= \text{if } m_j < \bar{m})$$

$$w \geq BR(Z)$$

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

In [3]:
```
"""
Author: Sebastian Graves

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:
    """
    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, ϑ) space for initial guess of equilibrium sets
        p_vec = np.zeros(self.N_a)
        w_vec = np.zeros(self.N_a)
        for i in range(self.N_a):
            p_vec[i] = self.Θ_vec[i]
            w_vec[i] = self.u_vec[i]/(1 - β)

        w_space = np.array([min(w_vec[~np.isinf(w_vec)]),
                            max(w_vec[~np.isinf(w_vec)])])
        p_space = np.array([0, max(p_vec[~np.isinf(w_vec)])])
        self.p_space = p_space

        # Set up hyperplane levels and gradients for iterations
        def SG_H_V(N, w_space, p_space):
            """
```

```
            This function  initializes the subgradients, hyperplane levels,
            and extreme points of the value set by choosing an appropriate
            origin and radius. It is based on a similar function in
↪QuantEcon's
            Games.jl
            """

            # First, create a unit circle. Want points placed on [0, 2π]
            inc = 2 * np.pi / N
            degrees = np.arange(0, 2 * np.pi, inc)

            # Points on circle
            H = np.zeros((N, 2))
            for i in range(N):
                x = degrees[i]
                H[i, 0] = np.cos(x)
                H[i, 1] = np.sin(x)

            # Then calculate origin and radius
            o = np.array([np.mean(w_space), np.mean(p_space)])
            r1 = max((max(w_space) - o[0])**2, (o[0] - min(w_space))**2)
            r2 = max((max(p_space) - o[1])**2, (o[1] - min(p_space))**2)
            r = np.sqrt(r1 + r2)

            # Now calculate vertices
            Z = np.zeros((2, N))
            for i in range(N):
                Z[0, i] = o[0] + r*H.T[0, i]
                Z[1, i] = o[1] + r*H.T[1, i]

            # Corresponding hyperplane levels
            C = np.zeros(N)
            for i in range(N):
                C[i] = np.dot(Z[:, i], H[i, :])

            return C, H, Z

        C, self.H, Z = SG_H_V(N_g, w_space, p_space)
        C = C.reshape(N_g, 1)
        self.c0_c, self.c0_s, self.c1_c, self.c1_s = np.copy(C), np.
↪copy(C), \
            np.copy(C), np.copy(C)
        self.z0_s, self.z0_c, self.z1_s, self.z1_c = np.copy(Z), np.
↪copy(Z), \
            np.copy(Z), np.copy(Z)

        self.w_bnds_s, self.w_bnds_c = (w_space[0], w_space[1]), \
            (w_space[0], w_space[1])
        self.p_bnds_s, self.p_bnds_c = (p_space[0], p_space[1]), \
            (p_space[0], p_space[1])

        # Create dictionaries to save equilibrium set for each iteration
        self.c_dic_s, self.c_dic_c = {}, {}
        self.c_dic_s[0], self.c_dic_c[0] = self.c0_s, self.c0_c

    def solve_worst_spe(self):
        """
        Method to solve for BR(Z). See p.449 of Chang (1998)
```

```
        """

        p_vec = np.full(self.N_a, np.nan)
        c = [1, 0]

        # Pre-compute constraints
        aineq_mbar = np.vstack((self.H, np.array([0, -self.β])))
        bineq_mbar = np.vstack((self.c0_s, 0))

        aineq = self.H
        bineq = self.c0_s
        aeq = [[0, -self.β]]

        for j in range(self.N_a):
            # Only try if consumption is possible
            if self.f_vec[j] > 0:
                # If m = mbar, use inequality constraint
                if self.A[j, 1] == self.mbar:
                    bineq_mbar[-1] = self.euler_vec[j]
                    res = linprog(c, A_ub=aineq_mbar, b_ub=bineq_mbar,
                                  bounds=(self.w_bnds_s, self.p_bnds_s))
                else:
                    beq = self.euler_vec[j]
                    res = linprog(c, A_ub=aineq, b_ub=bineq, A_eq=aeq,
b_eq=beq,
                                  bounds=(self.w_bnds_s, self.p_bnds_s))
                if res.status == 0:
                    p_vec[j] = self.u_vec[j] + self.β * res.x[0]

        # Max over h and min over other variables (see Chang (1998) p.449)
        self.br_z = np.nanmax(np.nanmin(p_vec.reshape(self.n_m, self.n_h),
0))

    def solve_subgradient(self):
        """
        Method to solve for E(Z). See p.449 of Chang (1998)
        """

        # Pre-compute constraints
        aineq_C_mbar = np.vstack((self.H, np.array([0, -self.β])))
        bineq_C_mbar = np.vstack((self.c0_c, 0))

        aineq_C = self.H
        bineq_C = self.c0_c
        aeq_C = [[0, -self.β]]

        aineq_S_mbar = np.vstack((np.vstack((self.H, np.array([0, -self.
β]))),
                                  np.array([-self.β, 0])))
        bineq_S_mbar = np.vstack((self.c0_s, np.zeros((2, 1))))

        aineq_S = np.vstack((self.H, np.array([-self.β, 0])))
        bineq_S = np.vstack((self.c0_s, 0))
        aeq_S = [[0, -self.β]]

        # Update maximal hyperplane level
        for i in range(self.N_g):
            c_a1a2_c, t_a1a2_c = np.full(self.N_a, -np.inf), \
```

```
                    np.zeros((self.N_a, 2))
                c_a1a2_s, t_a1a2_s = np.full(self.N_a, -np.inf), \
                    np.zeros((self.N_a, 2))

                c = [-self.H[i, 0], -self.H[i, 1]]

                for j in range(self.N_a):
                    # Only try if consumption is possible
                    if self.f_vec[j] > 0:

                        # COMPETITIVE EQUILIBRIA
                        # If m = mbar, use inequality constraint
                        if self.A[j, 1] == self.mbar:
                            bineq_C_mbar[-1] = self.euler_vec[j]
                         res = linprog(c, A_ub=aineq_C_mbar, b_ub=bineq_C_mbar,
                                        bounds=(self.w_bnds_c, self.p_bnds_c))
                        # If m < mbar, use equality constraint
                        else:
                            beq_C = self.euler_vec[j]
                           res = linprog(c, A_ub=aineq_C, b_ub=bineq_C, A_eq =
↪aeq_C,
                                        b_eq = beq_C, bounds=(self.w_bnds_c, \
                                              self.p_bnds_c))
                        if res.status == 0:
                            c_a1a2_c[j] = self.H[i, 0] * (self.u_vec[j] \
                            + self.β * res.x[0]) + self.H[i, 1] * self.Θ_vec[j]
                            t_a1a2_c[j] = res.x

                        # SUSTAINABLE EQUILIBRIA
                        # If m = mbar, use inequality constraint
                        if self.A[j, 1] == self.mbar:
                            bineq_S_mbar[-2] = self.euler_vec[j]
                            bineq_S_mbar[-1] = self.u_vec[j] - self.br_z
                         res = linprog(c, A_ub=aineq_S_mbar, b_ub=bineq_S_mbar,
                                        bounds=(self.w_bnds_s, self.p_bnds_s))
                        # If m < mbar, use equality constraint
                        else:
                            bineq_S[-1] = self.u_vec[j] - self.br_z
                            beq_S = self.euler_vec[j]
                           res = linprog(c, A_ub=aineq_S, b_ub=bineq_S, A_eq =
↪aeq_S,
                                        b_eq = beq_S, bounds=(self.w_bnds_s, \
                                              self.p_bnds_s))
                        if res.status == 0:
                            c_a1a2_s[j] = self.H[i, 0] * (self.u_vec[j] \
                              + self.β*res.x[0]) + self.H[i, 1] * self.Θ_vec[j]
                            t_a1a2_s[j] = res.x

                idx_c = np.where(c_a1a2_c == max(c_a1a2_c))[0][0]
                self.z1_c[:, i] = np.array([self.u_vec[idx_c]
                                            + self.β * t_a1a2_c[idx_c, 0],
                                            self.Θ_vec[idx_c]])

                idx_s = np.where(c_a1a2_s == max(c_a1a2_s))[0][0]
                self.z1_s[:, i] = np.array([self.u_vec[idx_s]
                                            + self.β * t_a1a2_s[idx_s, 0],
                                            self.Θ_vec[idx_s]])
```

```
        for i in range(self.N_g):
            self.c1_c[i] = np.dot(self.z1_c[:, i], self.H[i, :])
            self.c1_s[i] = np.dot(self.z1_s[:, i], self.H[i, :])

    def solve_sustainable(self, tol=1e-5, max_iter=250):
        """
      Method to solve for the competitive and sustainable equilibrium sets.
        """

        t = time.time()
        diff = tol + 1
        iters = 0

        print('### --------------- ###')
        print('Solving Chang Model Using Outer Hyperplane Approximation')
        print('### --------------- ### \n')

        print('Maximum difference when updating hyperplane levels:')

        while diff > tol and iters < max_iter:
            iters = iters + 1
            self.solve_worst_spe()
            self.solve_subgradient()
            diff = max(np.maximum(abs(self.c0_c - self.c1_c),
                       abs(self.c0_s - self.c1_s)))
            print(diff)

            # Update hyperplane levels
            self.c0_c, self.c0_s = np.copy(self.c1_c), np.copy(self.c1_s)

            # Update bounds for w and ϑ
            wmin_c, wmax_c = np.min(self.z1_c, axis=1)[0], \
                np.max(self.z1_c, axis=1)[0]
            pmin_c, pmax_c = np.min(self.z1_c, axis=1)[1], \
                np.max(self.z1_c, axis=1)[1]

            wmin_s, wmax_s = np.min(self.z1_s, axis=1)[0], \
                np.max(self.z1_s, axis=1)[0]
            pmin_S, pmax_S = np.min(self.z1_s, axis=1)[1], \
                np.max(self.z1_s, axis=1)[1]

          self.w_bnds_s, self.w_bnds_c = (wmin_s, wmax_s), (wmin_c, wmax_c)
          self.p_bnds_s, self.p_bnds_c = (pmin_S, pmax_S), (pmin_c, pmax_c)

            # Save iteration
           self.c_dic_c[iters], self.c_dic_s[iters] = np.copy(self.c1_c), \
                np.copy(self.c1_s)
            self.iters = iters

        elapsed = time.time() - t
        print('Convergence achieved after {} iterations and {} \
            seconds'.format(iters, round(elapsed, 2)))

    def solve_bellman(self, θ_min, θ_max, order, disp=False, tol=1e-7,
↪maxiters=100):
        """
        Continuous Method to solve the Bellman equation in section 25.3
        """
```

```
    mbar = self.mbar

    # Utility and production functions
    uc = lambda c: np.log(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

  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))

        # 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

    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)
```

103.4.1 Comparison of Sets

The set of (w, θ) associated with sustainable plans is smaller than the set of (w, θ) pairs associated with competitive equilibria, since the additional constraints associated with sustainability must also be satisfied.

Let's compute two examples, one with a low β, another with a higher β

```
In [4]: ch1 = ChangModel(β=0.3, mbar=30, h_min=0.9, h_max=2, n_h=8, n_m=35, N_g=10)
```

```
In [5]: 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]
```

```
↪---------------------------------------------------------------------------

        ValueError                                Traceback (most
↪recent call last)
```

```
      <ipython-input-5-ce0f3c9d3306> in <module>
  ----> 1 ch1.solve_sustainable()


      <ipython-input-3-ae015887b922> in solve_sustainable(self,
↪tol, max_iter)
      271             iters = iters + 1
      272             self.solve_worst_spe()
  --> 273             self.solve_subgradient()
      274             diff = max(np.maximum(abs(self.c0_c - self.
↪c1_c),
      275                       abs(self.c0_s - self.c1_s)))


      <ipython-input-3-ae015887b922> in solve_subgradient(self)
      233                         res = linprog(c, A_ub=aineq_S,
↪b_ub=bineq_S, A_eq = aeq_S,
      234                                       b_eq = beq_S,
↪bounds=(self.w_bnds_s, \
  --> 235                                           self.p_bnds_s))
      236                     if res.status == 0:
      237                       c_a1a2_s[j] = self.H[i, 0] * (self.
↪u_vec[j] \


      ~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_linprog.py in linprog(c, A_ub, b_ub, A_eq, b_eq, bounds, method,
↪callback, options, x0)
      539     x, fun, slack, con, status, message = _postprocess(
      540         x, c_o, A_ub_o, b_ub_o, A_eq_o, b_eq_o, bounds,
  --> 541         complete, undo, status, message, tol, iteration,
↪disp)
      542
      543     sol = {


      ~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_linprog_util.py in _postprocess(x, c, A_ub, b_ub, A_eq, b_eq,
↪bounds, complete, undo, status, message, tol, iteration, disp)
     1412     status, message = _check_result(
     1413         x, fun, status, slack, con,
  -> 1414         lb, ub, tol, message
     1415     )
     1416


      ~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_linprog_util.py in _check_result(x, fun, status, slack, con, lb,
↪ub, tol, message)
     1325         # nearly basic feasible solution. Postsolving can
↪make the solution
```

```
   1326         # basic, however, this solution is NOT optimal
-> 1327         raise ValueError(message)
   1328
   1329     return status, message


ValueError: The algorithm terminated successfully and
↪determined that the problem is infeasible.
```

The following plot shows both the set of w, θ pairs associated with competitive equilibria (in red) and the smaller set of w, θ pairs associated with sustainable plans (in blue).

```
In [6]: def plot_equilibria(ChangModel):
            """
            Method to plot both equilibrium sets
            """
            fig, ax = plt.subplots(figsize=(7, 5))

            ax.set_xlabel('w', fontsize=16)
            ax.set_ylabel(r"$\theta$", fontsize=18)

            poly_S = polytope.Polytope(ChangModel.H, ChangModel.c1_s)
            poly_C = polytope.Polytope(ChangModel.H, ChangModel.c1_c)
            ext_C = polytope.extreme(poly_C)
            ext_S = polytope.extreme(poly_S)

            ax.fill(ext_C[:, 0], ext_C[:, 1], 'r', zorder=-1)
            ax.fill(ext_S[:, 0], ext_S[:, 1], 'b', zorder=0)

            # 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

```
In [7]: 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)
```

```
In [8]: 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]
```

```
    
↪---------------------------------------------------------------------------

    ValueError                                Traceback (most
↪recent call last)

    <ipython-input-8-b1776dca964b> in <module>
  ----> 1 ch2.solve_sustainable()


    <ipython-input-3-ae015887b922> in solve_sustainable(self,
↪tol, max_iter)
    271             iters = iters + 1
    272             self.solve_worst_spe()
  --> 273             self.solve_subgradient()
    274             diff = max(np.maximum(abs(self.c0_c - self.
↪c1_c),
    275                        abs(self.c0_s - self.c1_s)))


    <ipython-input-3-ae015887b922> in solve_subgradient(self)
    233                         res = linprog(c, A_ub=aineq_S,
↪b_ub=bineq_S, A_eq = aeq_S,
    234                                       b_eq = beq_S,
↪bounds=(self.w_bnds_s, \
  --> 235                                           self.p_bnds_s))
    236                     if res.status == 0:
    237                       c_a1a2_s[j] = self.H[i, 0] * (self.
↪u_vec[j] \


    ~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_linprog.py in linprog(c, A_ub, b_ub, A_eq, b_eq, bounds, method,
↪callback, options, x0)
    539     x, fun, slack, con, status, message = _postprocess(
    540         x, c_o, A_ub_o, b_ub_o, A_eq_o, b_eq_o, bounds,
  --> 541         complete, undo, status, message, tol, iteration,
↪disp)
    542
    543     sol = {


    ~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_linprog_util.py in _postprocess(x, c, A_ub, b_ub, A_eq, b_eq,
↪bounds, complete, undo, status, message, tol, iteration, disp)
   1412     status, message = _check_result(
   1413         x, fun, status, slack, con,
  -> 1414         lb, ub, tol, message
   1415     )
```

```
      1416


       ~/anaconda3/lib/python3.7/site-packages/scipy/optimize/
↪_linprog_util.py in _check_result(x, fun, status, slack, con, lb,
↪ub, tol, message)
     1325         # nearly basic feasible solution. Postsolving can
↪make the solution
     1326         # basic, however, this solution is NOT optimal
   -> 1327         raise ValueError(message)
     1328
     1329     return status, message


       ValueError: The algorithm terminated successfully and
↪determined that the problem is infeasible.
```

Let's plot both sets

```
In [9]: plot_equilibria(ch2)
```

Evidently, the Ramsey plan is now sustainable.

Bibliography

[1] Dilip Abreu. On the theory of infinitely repeated games with discounting. *Econometrica*, 56:383–396, 1988.

[2] Dilip Abreu, David Pearce, and Ennio Stacchetti. Toward a theory of discounted repeated games with imperfect monitoring. *Econometrica*, 58(5):1041–1063, September 1990.

[3] Daron Acemoglu, Simon Johnson, and James A Robinson. The colonial origins of comparative development: An empirical investigation. *The American Economic Review*, 91(5):1369–1401, 2001.

[4] Daron Acemoglu and James A. Robinson. The political economy of the Kuznets curve. *Review of Development Economics*, 6(2):183–203, 2002.

[5] SeHyoun Ahn, Greg Kaplan, Benjamin Moll, Thomas Winberry, and Christian Wolf. When inequality matters for macro and macro matters for inequality. *NBER Macroeconomics Annual*, 32(1):1–75, 2018.

[6] S Rao Aiyagari. Uninsured Idiosyncratic Risk and Aggregate Saving. *The Quarterly Journal of Economics*, 109(3):659–684, 1994.

[7] 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.

[8] D. B. O. Anderson and J. B. Moore. *Optimal Filtering*. Dover Publications, 2005.

[9] E. W. Anderson, L. P. Hansen, E. R. McGrattan, and T. J. Sargent. Mechanics of Forming and Estimating Dynamic Linear Economies. In *Handbook of Computational Economics*. Elsevier, vol 1 edition, 1996.

[10] Cristina Arellano. Default risk and income fluctuations in emerging economies. *The American Economic Review*, pages 690–712, 2008.

[11] Papoulis Athanasios and S Unnikrishna Pillai. *Probability, random variables, and stochastic processes*. Mc-Graw Hill, 1991.

[12] 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.

[13] Robert L Axtell. Zipf distribution of us firm sizes. *science*, 293(5536):1818–1820, 2001.

[14] Robert J Barro. On the Determination of the Public Debt. *Journal of Political Economy*, 87(5):940–971, 1979.

[15] Robert J Barro. Determinants of democracy. *Journal of Political economy*, 107(S6):S158–S183, 1999.

[16] Robert J Barro and Rachel McCleary. Religion and economic growth. Technical report, National Bureau of Economic Research, 2003.

[17] Jess Benhabib and Alberto Bisin. Skewed wealth distributions: Theory and empirics. *Journal of Economic Literature*, 56(4):1261–91, 2018.

[18] Jess Benhabib, Alberto Bisin, and Shenghao Zhu. The wealth distribution in bewley economies with capital income risk. *Journal of Economic Theory*, 159:489–515, 2015.

[19] L M Benveniste and J A Scheinkman. On the Differentiability of the Value Function in Dynamic Models of Economics. *Econometrica*, 47(3):727–732, 1979.

[20] Dmitri Bertsekas. *Dynamic Programming and Stochastic Control*. Academic Press, New York, 1975.

[21] Truman Bewley. The permanent income hypothesis: A theoretical formulation. *Journal of Economic Theory*, 16(2):252–292, 1977.

[22] Truman F Bewley. Stationary monetary equilibrium with a continuum of independently fluctuating consumers. In Werner Hildenbran and Andreu Mas-Colell, editors, *Contributions to Mathematical Economics in Honor of Gerard Debreu*, pages 27–102. North-Holland, Amsterdam, 1986.

[23] 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.

[24] Anmol Bhandari, David Evans, Mikhail Golosov, and Thomas J Sargent. Inequality, business cycles, and monetary-fiscal policy. Technical report, National Bureau of Economic Research, 2018.

[25] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[26] Fischer Black and Robert Litterman. Global portfolio optimization. *Financial analysts journal*, 48(5):28–43, 1992.

[27] Dariusz Buraczewski, Ewa Damek, Thomas Mikosch, et al. *Stochastic models with power-law tails*. Springer, 2016.

[28] 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.

[29] Guillermo A. Calvo. On the time consistency of optimal policy in a monetary economy. *Econometrica*, 46(6):1411–1428, 1978.

[30] Andrew S Caplin. The variability of aggregate demand with (s, s) inventory policies. *Econometrica*, pages 1395–1409, 1985.

[31] Christopher D Carroll. A Theory of the Consumption Function, with and without Liquidity Constraints. *Journal of Economic Perspectives*, 15(3):23–45, 2001.

[32] Christopher D Carroll. The method of endogenous gridpoints for solving dynamic stochastic optimization problems. *Economics Letters*, 91(3):312–320, 2006.

[33] David Cass. Optimum growth in an aggregative model of capital accumulation. *Review of Economic Studies*, 32(3):233–240, 1965.

[34] Roberto Chang. Credible monetary policy in an infinite horizon model: Recursive approaches. *Journal of Economic Theory*, 81(2):431–461, 1998.

[35] Varadarajan V Chari and Patrick J Kehoe. Sustainable plans. *Journal of Political Economy*, pages 783–802, 1990.

[36] Ronald Harry Coase. The nature of the firm. *economica*, 4(16):386–405, 1937.

[37] Wilbur John Coleman. Solving the Stochastic Growth Model by Policy-Function Iteration. *Journal of Business & Economic Statistics*, 8(1):27–29, 1990.

[38] J. D. Cryer and K-S. Chan. *Time Series Analysis*. Springer, 2nd edition edition, 2008.

[39] Steven J Davis, R Jason Faberman, and John Haltiwanger. The flow approach to labor markets: New data sources, micro-macro links and the recent downturn. *Journal of Economic Perspectives*, 2006.

[40] Bruno de Finetti. La prevision: Ses lois logiques, ses sources subjectives. *Annales de l'Institute Henri Poincare'*, 7:1 – 68, 1937. English translation in Kyburg and Smokler (eds.), *Studies in Subjective Probability*, Wiley, New York, 1964.

[41] Angus Deaton. Saving and Liquidity Constraints. *Econometrica*, 59(5):1221–1248, 1991.

[42] Angus Deaton and Christina Paxson. Intertemporal Choice and Inequality. *Journal of Political Economy*, 102(3):437–467, 1994.

[43] Wouter J Den Haan. Comparison of solutions to the incomplete markets model with aggregate uncertainty. *Journal of Economic Dynamics and Control*, 34(1):4–27, 2010.

[44] 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.

[45] 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.

[46] 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.

[47] Ulrich Doraszelski and Mark Satterthwaite. Computable markov-perfect industry dynamics. *The RAND Journal of Economics*, 41(2):215–243, 2010.

[48] Y E Du, Ehud Lehrer, and A D Y Pauzner. Competitive economy as a ranking device over networks. submitted, 2013.

[49] R M Dudley. *Real Analysis and Probability*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2002.

[50] Timothy Dunne, Mark J Roberts, and Larry Samuelson. The growth and failure of us manufacturing plants. *The Quarterly Journal of Economics*, 104(4):671–698, 1989.

[51] Robert F Engle and Clive W J Granger. Co-integration and Error Correction: Representation, Estimation, and Testing. *Econometrica*, 55(2):251–276, 1987.

[52] Richard Ericson and Ariel Pakes. Markov-perfect industry dynamics: A framework for empirical work. *The Review of Economic Studies*, 62(1):53–82, 1995.

[53] David S Evans. The relationship between firm growth, size, and age: Estimates for 100 manufacturing industries. *The Journal of Industrial Economics*, pages 567–581, 1987.

[54] G W Evans and S Honkapohja. *Learning and Expectations in Macroeconomics*. Frontiers of Economic Research. Princeton University Press, 2001.

[55] Pablo Fajgelbaum, Edouard Schaal, and Mathieu Taschereau-Dumouchel. Uncertainty traps. Technical report, National Bureau of Economic Research, 2015.

[56] M. Friedman. *A Theory of the Consumption Function*. Princeton University Press, 1956.

[57] Milton Friedman and Rose D Friedman. *Two Lucky People*. University of Chicago Press, 1998.

[58] Yoshi Fujiwara, Corrado Di Guilmi, Hideaki Aoyama, Mauro Gallegati, and Wataru Souma. Do pareto–zipf and gibrat laws hold true? an analysis with european firms. *Physica A: Statistical Mechanics and its Applications*, 335(1-2):197–216, 2004.

[59] Xavier Gabaix. Power laws in economics: An introduction. *Journal of Economic Perspectives*, 30(1):185–206, 2016.

[60] David Gale. *The theory of linear economic models*. University of Chicago press, 1989.

[61] 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.

[62] Robert Gibrat. *Les inégalités économiques: Applications d'une loi nouvelle, la loi de l'effet proportionnel*. PhD thesis, Recueil Sirey, 1931.

[63] Edward Glaeser, Jose Scheinkman, and Andrei Shleifer. The injustice of inequality. *Journal of Monetary Economics*, 50(1):199–222, 2003.

[64] Geoffrey J Gordon. Stable function approximation in dynamic programming. In *Machine Learning Proceedings 1995*, pages 261–268. Elsevier, 1995.

[65] Olle Häggström. *Finite Markov chains and algorithmic applications*, volume 52. Cambridge University Press, 2002.

[66] Bronwyn H Hall. The relationship between firm size and firm growth in the us manufacturing sector. *The Journal of Industrial Economics*, pages 583–606, 1987.

[67] Robert E Hall. Stochastic Implications of the Life Cycle-Permanent Income Hypothesis: Theory and Evidence. *Journal of Political Economy*, 86(6):971–987, 1978.

[68] Robert E Hall and Frederic S Mishkin. The Sensitivity of Consumption to Transitory Income: Estimates from Panel Data on Households. *National Bureau of Economic Research Working Paper Series*, No. 505, 1982.

[69] Michael J Hamburger, Gerald L Thompson, and Roman L Weil. Computation of expansion rates for the generalized von neumann model of an expanding economy. *Econometrica, Journal of the Econometric Society*, pages 542–547, 1967.

[70] James D Hamilton. What's real about the business cycle? *Federal Reserve Bank of St. Louis Review*, (July-August):435–452, 2005.

[71] L P Hansen and T J Sargent. *Robustness*. Princeton University Press, 2008.

[72] L P Hansen and T J Sargent. *Recursive Models of Dynamic Linear Economies*. The Gorman Lectures in Economics. Princeton University Press, 2013.

[73] Lars Peter Hansen and Scott F Richard. The Role of Conditioning Information in Deducing Testable. *Econometrica*, 55(3):587–613, May 1987.

[74] 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.

[75] Lars Peter Hansen and Thomas J Sargent. Wanting robustness in macroeconomics. *Manuscript, Department of Economics, Stanford University.*, 4, 2000.

[76] Lars Peter Hansen and Thomas J. Sargent. Robust control and model uncertainty. *American Economic Review*, 91(2):60–66, 2001.

[77] Lars Peter Hansen and Thomas J Sargent. *Robustness*. Princeton university press, 2008.

[78] Lars Peter Hansen and Thomas J. Sargent. *Recursive Linear Models of Dynamic Economics*. Princeton University Press, Princeton, New Jersey, 2013.

[79] Lars Peter Hansen and José A Scheinkman. Long-term risk: An operator approach. *Econometrica*, 77(1):177–234, 2009.

[80] J. Michael Harrison and David M. Kreps. Speculative investor behavior in a stock market with heterogeneous expectations. *The Quarterly Journal of Economics*, 92(2):323–336, 1978.

[81] J. Michael Harrison and David M. Kreps. Martingales and arbitrage in multiperiod securities markets. *Journal of Economic Theory*, 20(3):381–408, June 1979.

[82] John Heaton and Deborah J Lucas. Evaluating the effects of incomplete markets on risk sharing and asset pricing. *Journal of Political Economy*, pages 443–487, 1996.

[83] Elhanan Helpman and Paul Krugman. *Market structure and international trade*. MIT Press Cambridge, 1985.

[84] 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.

[85] Hugo A Hopenhayn. Entry, exit, and firm dynamics in long run equilibrium. *Econometrica: Journal of the Econometric Society*, pages 1127–1150, 1992.

[86] Hugo A Hopenhayn and Edward C Prescott. Stochastic Monotonicity and Stationary Distributions for Dynamic Economies. *Econometrica*, 60(6):1387–1406, 1992.

[87] Hugo A Hopenhayn and Richard Rogerson. Job Turnover and Policy Evaluation: A General Equilibrium Analysis. *Journal of Political Economy*, 101(5):915–938, 1993.

[88] Mark Huggett. The risk-free rate in heterogeneous-agent incomplete-insurance economies. *Journal of Economic Dynamics and Control*, 17(5-6):953–969, 1993.

[89] K Jänich. *Linear Algebra*. Springer Undergraduate Texts in Mathematics and Technology. Springer, 1994.

[90] Robert J. Shiller John Y. Campbell. The Dividend-Price Ratio and Expectations of Future Dividends and Discount Factors. *Review of Financial Studies*, 1(3):195–228, 1988.

[91] Boyan Jovanovic. Firm-specific capital and turnover. *Journal of Political Economy*, 87(6):1246–1260, 1979.

[92] K L Judd. Cournot versus bertrand: A dynamic resolution. Technical report, Hoover Institution, Stanford University, 1990.

[93] Kenneth L Judd. On the performance of patents. *Econometrica*, pages 567–585, 1985.

[94] Kenneth L. Judd, Sevin Yeltekin, and James Conklin. Computing Supergame Equilibria. *Econometrica*, 71(4):1239–1254, 07 2003.

[95] Takashi Kamihigashi. Elementary results on solutions to the bellman equation of dynamic programming: existence, uniqueness, and convergence. Technical report, Kobe University, 2012.

[96] John G Kemeny, Oskar Morgenstern, and Gerald L Thompson. A generalization of the von neumann model of an expanding economy. *Econometrica, Journal of the Econometric Society*, pages 115–135, 1956.

[97] 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.

[98] Illenin Kondo, Logan T Lewis, and Andrea Stella. On the us firm and establishment size distributions. Technical report, SSRN, 2018.

[99] Tjalling C. Koopmans. On the concept of optimal economic growth. In Tjalling C. Koopmans, editor, *The Economic Approach to Development Planning*, page 225–287. Chicago, 1965.

[100] David M. Kreps. *Notes on the Theory of Choice*. Westview Press, Boulder, Colorado, 1988.

[101] Moritz Kuhn. Recursive Equilibria In An Aiyagari-Style Economy With Permanent Income Shocks. *International Economic Review*, 54:807–835, 2013.

[102] 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.

[103] A Lasota and M C MacKey. *Chaos, Fractals, and Noise: Stochastic Aspects of Dynamics*. Applied Mathematical Sciences. Springer-Verlag, 1994.

[104] Edward E Leamer. *Specification searches: Ad hoc inference with nonexperimental data*, volume 53. John Wiley & Sons Incorporated, 1978.

[105] Martin Lettau and Sydney Ludvigson. Consumption, Aggregate Wealth, and Expected Stock Returns. *Journal of Finance*, 56(3):815–849, 06 2001.

[106] Martin Lettau and Sydney C. Ludvigson. Understanding Trend and Cycle in Asset Values: Reevaluating the Wealth Effect on Consumption. *American Economic Review*, 94(1):276–299, March 2004.

[107] David Levhari and Leonard J Mirman. The great fish war: an example using a dynamic cournot-nash solution. *The Bell Journal of Economics*, pages 322–334, 1980.

[108] L Ljungqvist and T J Sargent. *Recursive Macroeconomic Theory*. MIT Press, 4 edition, 2018.

[109] Robert E Lucas, Jr. Asset prices in an exchange economy. *Econometrica: Journal of the Econometric Society*, 46(6):1429–1445, 1978.

[110] Robert E Lucas, Jr. and Edward C Prescott. Investment under uncertainty. *Econometrica: Journal of the Econometric Society*, pages 659–681, 1971.

[111] 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.

[112] Benoit Mandelbrot. The variation of certain speculative prices. *The Journal of Business*, 36(4):394–419, 1963.

[113] Albert Marcet and Thomas J Sargent. Convergence of Least-Squares Learning in Environments with Hidden State Variables and Private Information. *Journal of Political Economy*, 97(6):1306–1322, 1989.

[114] V Filipe Martins-da Rocha and Yiannis Vailakis. Existence and Uniqueness of a Fixed Point for Local Contractions. *Econometrica*, 78(3):1127–1141, 2010.

[115] A Mas-Colell, M D Whinston, and J R Green. *Microeconomic Theory*, volume 1. Oxford University Press, 1995.

[116] J J McCall. Economics of Information and Job Search. *The Quarterly Journal of Economics*, 84(1):113–126, 1970.

[117] S P Meyn and R L Tweedie. *Markov Chains and Stochastic Stability.* Cambridge University Press, 2009.

[118] Mario J Miranda and P L Fackler. *Applied Computational Economics and Finance.* Cambridge: MIT Press, 2002.

[119] F. Modigliani and R. Brumberg. Utility analysis and the consumption function: An interpretation of cross-section data. In K.K Kurihara, editor, *Post-Keynesian Economics.* 1954.

[120] John F Muth. Optimal properties of exponentially weighted forecasts. *Journal of the american statistical association*, 55(290):299–306, 1960.

[121] Derek Neal. The Complexity of Job Mobility among Young Men. *Journal of Labor Economics*, 17(2):237–261, 1999.

[122] Y Nishiyama, S Osada, and K Morimune. Estimation and testing for rank size rule regression under pareto distribution. In *Proceedings of the International Environmental Modelling and Software Society iEMSs 2004 International Conference.* Citeseer, 2004.

[123] Sophocles J Orfanidis. *Optimum Signal Processing: An Introduction.* McGraw Hill Publishing, New York, New York, 1988.

[124] Jenő Pál and John Stachurski. Fitted value function iteration with probability one contractions. *Journal of Economic Dynamics and Control*, 37(1):251–264, 2013.

[125] Jonathan A Parker. The Reaction of Household Consumption to Predictable Changes in Social Security Taxes. *American Economic Review*, 89(4):959–973, 1999.

[126] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming.* John Wiley & Sons, 2005.

[127] Guillaume Rabault. When do borrowing constraints bind? Some new results on the income fluctuation problem. *Journal of Economic Dynamics and Control*, 26(2):217–245, 2002.

[128] Svetlozar Todorov Rachev. *Handbook of heavy tailed distributions in finance: Handbooks in finance*, volume 1. Elsevier, 2003.

[129] F. P. Ramsey. A Contribution to the theory of taxation. *Economic Journal*, 37(145):47–61, 1927.

[130] Kevin L Reffett. Production-based asset pricing in monetary economies with transactions costs. *Economica*, pages 427–443, 1996.

[131] Michael Reiter. Solving heterogeneous-agent models by projection and perturbation. *Journal of Economic Dynamics and Control*, 33(3):649–665, 2009.

[132] Steven Roman. *Advanced linear algebra*, volume 3. Springer, 2005.

[133] Sherwin Rosen, Kevin M Murphy, and Jose A Scheinkman. Cattle cycles. *Journal of Political Economy*, 102(3):468–492, 1994.

[134] Y. A. Rozanov. *Stationary Random Processes.* Holden-Day, San Francisco, 1967.

[135] Hernán D Rozenfeld, Diego Rybski, Xavier Gabaix, and Hernán A Makse. The area and population of cities: New insights from a different perspective on cities. *American Economic Review*, 101(5):2205–25, 2011.

[136] John Rust. Numerical dynamic programming in economics. *Handbook of computational economics*, 1:619–729, 1996.

[137] Stephen P Ryan. The costs of environmental regulation in a concentrated industry. *Econometrica*, 80(3):1019–1061, 2012.

[138] Jaewoo Ryoo and Sherwin Rosen. The engineering labor market. *Journal of political economy*, 112(S1):S110–S140, 2004.

[139] Paul A. Samuelson. Interactions between the multiplier analysis and the principle of acceleration. *Review of Economic Studies*, 21(2):75–78, 1939.

[140] Thomas Sargent, Lars Peter Hansen, and Will Roberts. Observable implications of present value budget balance. In *Rational Expectations Econometrics.* Westview Press, 1991.

[141] Thomas J Sargent. The Demand for Money During Hyperinflations under Rational Expectations: I. *International Economic Review*, 18(1):59–82, February 1977.

[142] Thomas J Sargent. *Macroeconomic Theory.* Academic Press, New York, 2nd edition, 1987.

[143] Jack Schechtman and Vera L S Escudero. Some results on an income fluctuation problem. *Journal of Economic Theory*, 16(2):151–166, 1977.

[144] Jose A. Scheinkman. *Speculation, Trading, and Bubbles.* Columbia University Press, New York, 2014.

[145] Thomas C Schelling. Models of Segregation. *American Economic Review*, 59(2):488–493, 1969.

[146] Christian Schluter and Mark Trede. Size distributions reconsidered. *Econometric Reviews*, 38(6):695–710, 2019.

[147] A N Shiriaev. *Probability*. Graduate texts in mathematics. Springer. Springer, 2nd edition, 1995.

[148] John Stachurski. Continuous state dynamic programming via nonexpansive approximation. *Computational Economics*, 31(2):141–160, 2008.

[149] N L Stokey, R E Lucas, and E C Prescott. *Recursive Methods in Economic Dynamics*. Harvard University Press, 1989.

[150] Nancy L Stokey. Reputation and time consistency. *The American Economic Review*, pages 134–139, 1989.

[151] Nancy L. Stokey. Credible public policy. *Journal of Economic Dynamics and Control*, 15(4):627–656, October 1991.

[152] Kjetil Storesletten, Christopher I Telmer, and Amir Yaron. Consumption and risk sharing over the life cycle. *Journal of Monetary Economics*, 51(3):609–633, 2004.

[153] R K Sundaram. *A First Course in Optimization Theory*. Cambridge University Press, 1996.

[154] 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, March 2009.

[155] Lars EO Svensson, Noah Williams, et al. Optimal monetary policy under uncertainty: A markov jump-linear-quadratic approach. *Federal Reserve Bank of St. Louis Review*, 90(4):275–293, 2008.

[156] George Tauchen. Finite state markov-chain approximations to univariate and vector autoregressions. *Economics Letters*, 20(2):177–181, 1986.

[157] Daniel Treisman. Russia's billionaires. *The American Economic Review*, 106(5):236–241, 2016.

[158] Ngo Van Long. Dynamic games in the economics of natural resources: a survey. *Dynamic Games and Applications*, 1(1):115–148, 2011.

[159] Pareto Vilfredo. Cours d'économie politique. *Rouge, Lausanne*, 2, 1896.

[160] John von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische annalen*, 100(1):295–320, 1928.

[161] John von Neumann. Uber ein okonomsiches gleichungssystem und eine verallgemeinering des browerschen fixpunktsatzes. In *Erge. Math. Kolloq.*, volume 8, pages 73–83, 1937.

[162] Abraham Wald. *Sequential Analysis*. John Wiley and Sons, New York, 1947.

[163] Peter Whittle. *Prediction and regulation by linear least-square methods*. English Univ. Press, 1963.

[164] Peter Whittle. *Prediction and Regulation by Linear Least Squares Methods.* University of Minnesota Press, Minneapolis, Minnesota, 2nd edition, 1983.

[165] Jeffrey M Wooldridge. *Introductory econometrics: A modern approach.* Nelson Education, 2015.

[166] G Alastair Young and Richard L Smith. *Essentials of statistical inference.* Cambridge University Press, 2005.