This pdf presents a series of lectures on quantitative economic modeling, designed and written by Thomas J. Sargent and John Stachurski. The primary programming languages are Python and Julia. You can send feedback to the authors via contact@quantecon.org.

Note: You are currently viewing an automatically generated pdf version of our online lectures, which are located at

https://lectures.quantecon.org

Please visit the website for more information on the aims and scope of the lectures and the two language options (Julia or Python).

Due to automatic generation of this pdf, presentation quality is likely to be lower than that of the website.
## CONTENTS

1 **Introduction to Python** ................................................. 1  
1.1 About Python .............................................................. 1  
1.2 Setting up Your Python Environment ................................. 13  
1.3 An Introductory Example ............................................... 37  
1.4 Python Essentials ....................................................... 59  
1.5 OOP I: Introduction to Object Oriented Programming .......... 78  

2 **The Scientific Libraries** .............................................. 85  
2.1 NumPy ................................................................. 85  
2.2 Matplotlib ............................................................. 105  
2.3 SciPy ................................................................. 116  
2.4 Numba ................................................................. 127  
2.5 Other Scientific Libraries ............................................ 139  

3 **Advanced Python Programming** .................................... 149  
3.1 Writing Good Code .................................................... 149  
3.2 OOP II: Building Classes ............................................ 157  
3.3 OOP III: The Samuelson Accelerator ............................... 174  
3.4 More Language Features ............................................. 209  
3.5 Debugging ............................................................. 242  

4 **Data and Empirics** ..................................................... 249  
4.1 Pandas ................................................................. 249  
4.2 Pandas for Panel Data ................................................. 265  
4.3 Linear Regression in Python ......................................... 287  
4.4 Maximum Likelihood Estimation .................................... 304  

5 **Tools and Techniques** ................................................ 329  
5.1 Linear Algebra ......................................................... 329  
5.2 Orthogonal Projections and Their Applications ................. 353  
5.3 LLN and CLT .......................................................... 370  
5.4 Linear State Space Models .......................................... 391  
5.5 Finite Markov Chains ................................................ 417  
5.6 Continuous State Markov Chains ................................... 442  
5.7 A First Look at the Kalman Filter ................................... 465
### 6 Dynamic Programming

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Shortest Paths</td>
<td>485</td>
</tr>
<tr>
<td>6.2 Job Search I: The McCall Search Model</td>
<td>493</td>
</tr>
<tr>
<td>6.3 Job Search II: Search and Separation</td>
<td>506</td>
</tr>
<tr>
<td>6.4 A Problem that Stumped Milton Friedman</td>
<td>518</td>
</tr>
<tr>
<td>6.5 Job Search III: Search with Learning</td>
<td>538</td>
</tr>
<tr>
<td>6.6 Job Search IV: Modeling Career Choice</td>
<td>556</td>
</tr>
<tr>
<td>6.7 Job Search V: On-the-Job Search</td>
<td>570</td>
</tr>
<tr>
<td>6.8 Optimal Growth I: The Stochastic Optimal Growth Model</td>
<td>583</td>
</tr>
<tr>
<td>6.9 Optimal Growth II: Time Iteration</td>
<td>602</td>
</tr>
<tr>
<td>6.10 Optimal Growth III: The Endogenous Grid Method</td>
<td>621</td>
</tr>
<tr>
<td>6.11 LQ Dynamic Programming Problems</td>
<td>630</td>
</tr>
<tr>
<td>6.12 Optimal Savings I: The Permanent Income Model</td>
<td>662</td>
</tr>
<tr>
<td>6.13 Optimal Savings II: LQ Techniques</td>
<td>679</td>
</tr>
<tr>
<td>6.14 Consumption and Tax Smoothing with Complete and Incomplete Markets</td>
<td>697</td>
</tr>
<tr>
<td>6.15 Optimal Savings III: Occasionally Binding Constraints</td>
<td>716</td>
</tr>
<tr>
<td>6.16 Robustness</td>
<td>736</td>
</tr>
<tr>
<td>6.17 Discrete State Dynamic Programming</td>
<td>755</td>
</tr>
</tbody>
</table>

### 7 Multiple Agent Models

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Schellings Segregation Model</td>
<td>779</td>
</tr>
<tr>
<td>7.2 A Lake Model of Employment and Unemployment</td>
<td>791</td>
</tr>
<tr>
<td>7.3 Rational Expectations Equilibrium</td>
<td>819</td>
</tr>
<tr>
<td>7.4 Markov Perfect Equilibrium</td>
<td>833</td>
</tr>
<tr>
<td>7.5 Robust Markov Perfect Equilibrium</td>
<td>851</td>
</tr>
<tr>
<td>7.6 Asset Pricing I: Finite State Models</td>
<td>871</td>
</tr>
<tr>
<td>7.7 Asset Pricing II: The Lucas Asset Pricing Model</td>
<td>892</td>
</tr>
<tr>
<td>7.8 Asset Pricing III: Incomplete Markets</td>
<td>904</td>
</tr>
<tr>
<td>7.9 Uncertainty Traps</td>
<td>916</td>
</tr>
<tr>
<td>7.10 The Aiyagari Model</td>
<td>930</td>
</tr>
<tr>
<td>7.11 Default Risk and Income Fluctuations</td>
<td>938</td>
</tr>
<tr>
<td>7.12 Globalization and Cycles</td>
<td>960</td>
</tr>
</tbody>
</table>

### 8 Time Series Models

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Covariance Stationary Processes</td>
<td>979</td>
</tr>
<tr>
<td>8.2 Estimation of Spectra</td>
<td>997</td>
</tr>
<tr>
<td>8.3 Additive Functionals</td>
<td>1012</td>
</tr>
<tr>
<td>8.4 Multiplicative Functionals</td>
<td>1030</td>
</tr>
<tr>
<td>8.5 Classical Control with Linear Algebra</td>
<td>1051</td>
</tr>
<tr>
<td>8.6 Classical Filtering With Linear Algebra</td>
<td>1075</td>
</tr>
</tbody>
</table>

### 9 Dynamic Programming Squared

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Dynamic Stackelberg Problems</td>
<td>1097</td>
</tr>
<tr>
<td>9.2 Ramsey plans, Time Inconsistency, Sustainable Plans</td>
<td>1113</td>
</tr>
<tr>
<td>9.3 Optimal Taxation in an LQ Economy</td>
<td>1138</td>
</tr>
<tr>
<td>9.4 Optimal Taxation with State-Contingent Debt</td>
<td>1159</td>
</tr>
<tr>
<td>9.5 Optimal Taxation without State-Contingent Debt</td>
<td>1193</td>
</tr>
<tr>
<td>9.6 Fluctuating Interest Rates Deliver Fiscal Insurance</td>
<td>1220</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>9.7 Fiscal Risk and Government Debt</td>
<td>1246</td>
</tr>
<tr>
<td>9.8 Competitive Equilibria of Chang Model</td>
<td>1272</td>
</tr>
<tr>
<td>9.9 Credible Government Policies in Chang Model</td>
<td>1303</td>
</tr>
<tr>
<td>10 Quantitative Economics</td>
<td>1329</td>
</tr>
<tr>
<td>11 Tools and Techniques</td>
<td>1331</td>
</tr>
<tr>
<td>12 Dynamic Programming</td>
<td>1333</td>
</tr>
<tr>
<td>13 Multiple Agent Models</td>
<td>1335</td>
</tr>
<tr>
<td>14 Quantitative Economics</td>
<td>1337</td>
</tr>
<tr>
<td>15 Tools and Techniques</td>
<td>1339</td>
</tr>
<tr>
<td>16 Dynamic Programming</td>
<td>1341</td>
</tr>
<tr>
<td>17 Multiple Agent Models</td>
<td>1343</td>
</tr>
<tr>
<td>18 Time Series Models</td>
<td>1345</td>
</tr>
<tr>
<td>Bibliography</td>
<td>1347</td>
</tr>
</tbody>
</table>
CHAPTER ONE

INTRODUCTION TO PYTHON

This first part of the course provides a relatively fast-paced introduction to the Python programming language.

1.1 About Python

Contents

• About Python
  – Overview
  – What's Python?
  – Scientific Programming
  – Learn More

1.1.1 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.

1.1.2 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.

**Common Uses**

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

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

Used extensively by Internet service and high tech companies such as

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

Often used to teach computer science and programming

For reasons we will discuss, Python is particularly popular within the scientific community

- academia, NASA, CERN, Wall St., etc., etc.

**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 seek 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

**Features**

Python is a high level language suitable for rapid development.

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

Other features:

- A multiparadigm language, in that multiple programming styles are supported (procedural, object-oriented, functional, etc.)
- Interpreted rather than compiled

**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 its 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 elegant design.

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

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

### 1.1.3 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
- etc., etc.

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.

### 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:

```python
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:
The number you see here might vary slightly but its 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, lets calculate \( \int_{-2}^{2} \phi(z) \, dz \) where \( \phi \) is the standard normal density

```python
from scipy.stats import norm
from scipy.integrate import quad

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

0.9544997361036417

SciPy includes many of the standard routines used in
- linear algebra
- integration
- interpolation
- optimization
- distributions and random number generation
- signal processing
- etc., etc.

**Graphics**

The most popular and comprehensive Python library for creating figures and graphs is Matplotlib
- Plots, histograms, contour images, 3D, bar charts, etc., 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

### 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:

```python
from sympy import Symbol

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

3*x + y
```

We can manipulate expressions:

```python
expression = (x + y)**2
expression.expand()

x**2 + 2*x*y + y**2
```

solve polynomials:

```python
from sympy import solve

solve(x**2 + x + 2)

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

and calculate limits, derivatives and integrals:

```python
from sympy import limit, sin, diff

limit(1 / x, x, 0)

oo

limit(sin(x) / x, x, 0)

1

diff(sin(x), x)
```
The beauty of importing this functionality into Python is that we are working within a fully fledged pro-
gramming language
Can easily create tables of derivatives, generate LaTeX output, add it to figures, etc., etc.

**Statistics**

Pythons 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 is a simple example, using some fake data

```python
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)
```

<table>
<thead>
<tr>
<th></th>
<th>price</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010-12-28</td>
<td>0.471435</td>
<td>-1.190976</td>
</tr>
<tr>
<td>2010-12-29</td>
<td>1.432707</td>
<td>-0.312652</td>
</tr>
<tr>
<td>2010-12-30</td>
<td>-0.720589</td>
<td>0.887163</td>
</tr>
<tr>
<td>2010-12-31</td>
<td>0.859588</td>
<td>-0.636524</td>
</tr>
<tr>
<td>2011-01-01</td>
<td>0.015696</td>
<td>-2.242685</td>
</tr>
</tbody>
</table>

```python
df.mean()
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>0.411768</td>
</tr>
<tr>
<td>weight</td>
<td>-0.699135</td>
</tr>
<tr>
<td>dtype:</td>
<td>float64</td>
</tr>
</tbody>
</table>

**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**

### Networks and Graphs

Python has many libraries for studying graphs.

One well-known example is **NetworkX**

- Standard graph algorithms for analyzing network structure, etc.
- Plotting routines
- etc., etc.

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

```python
import networkx as nx
import matplotlib.pyplot as plt
np.random.seed(1234)

# Generate 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')

dists = [(x - 0.5)**2 + (y - 0.5)**2 for x, y in list(pos.values())]
cmp = np.argmin(dists)
ncenter = p.keys()

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()
xn.draw_networkx_edges(G, pos, alpha=0.4)
xn.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()
```
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

Parallel Processing

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

- Parallel computing through IPython clusters
- The Starcluster interface to Amazons EC2
- GPU programming through PyCuda, PyOpenCL, Theano or similar
Other Developments

There are many other interesting developments with scientific programming in Python

Some representative examples include

- **Jupyter**  Python in your browser with code cells, embedded images, etc.
- **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

1.1.4 Learn More

- Browse some Python projects on [GitHub](https://github.com)
- Have a look at some of the Jupyter notebooks people have shared on various scientific topics
- Visit the [Python Package Index](https://pypi.org)
- View some of the question people are asking about Python on [Stackoverflow](https://stackoverflow.com)
- Keep up to date on what's happening in the Python community with the [Python subreddit](https://www.reddit.com/r/Python)

1.2 Setting up Your Python Environment

### Contents

- **Setting up Your Python Environment**
  - Overview
  - Anaconda
  - Jupyter Notebooks
  - QuantEcon.py
  - Keeping Software up to Date
  - Working with Files
  - Editors and IDEs
  - Exercises
1.2.1 Overview

In this lecture you will learn how to

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

1.2.2 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 free Python distribution that contains

1. the core Python language and
2. 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!

Installing Anaconda

Installing Anaconda is straightforward: 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
- Otherwise you can accept all of the defaults
Get a Modern Browser

Well be using your browser to interact with Python, so now might be a good time to
1. update your browser, or
2. install a free modern browser such as Chrome or Firefox

1.2.3 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 possibilities, Jupyter is fast turning into a major player in the scientific computing ecosystem

Here’s an image of showing execution of some code (borrowed from here) in a Jupyter notebook
You can find a nice example of the kinds of things you can do in a Jupyter notebook (such as include maths and text) here

Further examples can be found at QuantEcons notebook archive or the NB viewer site

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

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 (click to enlarge)

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 (click to enlarge)

![Jupyter dashboard](image)

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 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

**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*
As a result, you can type in Python code and it will appear in the cell
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
• Whatever you type appears as is in that cell

2. Command mode
• The green border is replaced by a grey border
• Key strokes are interpreted as commands for example, typing \texttt{b} adds a new cell below the current one

To switch to
• command mode from edit mode, hit the \texttt{Esc} key or \texttt{Ctrl-M}
• edit mode from command mode, hit \texttt{Enter} or click in a cell

The modal behavior of the Jupyter notebook is a little tricky at first but very efficient when you get used to it

\section*{User Interface Tour}

At this stage we recommend you take your time to
• look at the various options in the menus and see what they do
• take the user interface tour, which can be accessed through the help menu

\section*{Inserting unicode (e.g., Greek letters)}

Python 3 introduced support for \textit{unicode characters}, allowing the use of characters such as $\alpha$ and $\beta$ in your code

Unicode characters can be typed quickly in Jupyter using the \texttt{tab} key

Try creating a new code cell and typing $\backslash \alpha$, then hitting the \texttt{tab} key on your keyboard

\section*{A Test Program}

Lets run a test program

Here is an arbitrary program we can use: \url{http://matplotlib.org/1.4.1/examples/pie_and_polar_charts/polar_bar_demo.html}

On that page you will see the following code

\begin{verbatim}
import numpy as np
import matplotlib.pyplot as plt

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)
\end{verbatim}
ax = plt.subplot(111, polar=True)
bars = ax.bar(θ, radii, width=width, bottom=0.0)

# Use custom colors and opacity
for r, bar in zip(radii, bars):
    bar.set_facecolor(plt.cm.jet(r / 10.))
    bar.set_alpha(0.5)

plt.show()

Dont 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 into a cell in the notebook.

You should see something like this.
(In older versions of Jupyter you might need to add the command `%matplotlib inline` before you generate the figure)

**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 well 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 this attributes of np using the Tab key

For example, here we type `np.ran` and hit Tab (click to enlarge)

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 in 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

1.2. Setting up Your Python Environment
For example, here we enter a mixture of plain text and LaTeX instead of code:

\[
\theta = np.linspace(0.0, 2 \cdot np.pi, N, \text{endpoint=False})
\]
\[
radii = 10 \cdot np.random.rand(N)
\]
\[
width = np.pi / 4 \cdot np.random.rand(N)
\]
\[
ax = plt.subplot(111, polar=True)
bars = ax.bar(theta, radii, width=width, bottom=0.0)
\]
\[
# Use custom colors and opacity
for r, bar in zip(radii, bars):
    bar.set_facecolor(plt.cm.jet(r / 10.))
    bar.set_alpha(0.5)
\]
\[
plt.show()
\]

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

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

Now we \texttt{Shift+Enter} to produce this:
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.
1.2.4 QuantEcon.py

In these lectures we will make extensive use of code from the QuantEcon organization.

On the Python side we will be using the QuantEcon.py version.

This code has been organized into a Python package.

- A Python package is a software library that has been bundled for distribution.
- Hosted Python packages can be found through channels like Anaconda and PyPi.

You can install QuantEcon.py by starting Jupyter and typing

```
!pip install 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.

**Note:** The QuantEcon.py package can also be installed using conda by:

```
conda config --add channels conda-forge
conda install quantecon
```

1.2.5 Keeping Software up to Date

For these lectures to run without error you need to keep your software up to date.

**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.

**Updating QuantEcon.py**

Open up a terminal and type

```
pip install --upgrade quantecon
```
Or open up Jupyter and type the same thing in a notebook cell with ! in front of it

### 1.2.6 Working with Files

How does one run a locally saved Python file?
There are a number of ways to do this but let’s focus on methods using Jupyter notebooks

**Option 1: Copy and Paste**

The steps are:

1. Navigate to your file with your mouse / trackpad using a file browser
2. Click on your file to open it with a text editor
3. Copy and paste into a cell and Shift-Enter

**Method 2: Run**

Using the `run` command is often easier than copy and paste

- For example, `%run test.py` will run the file `test.py`

(You might find that the `%` is unnecessary use `%automagic` to toggle the need for `%`)

Note that Jupyter only looks for `test.py` in the present working directory (PWD)

If `test.py` isn’t in that directory, you will get an error

Let’s look at a successful example, where we run a file `test.py` with contents:

```python
for i in range(5):
    print('foobar')
```

Here’s the notebook (click to enlarge)
Here

- **pwd** asks Jupyter to show the PWD (or `%pwd` see the comment about automagic above)
  - This is where Jupyter is going to look for files to run
  - Your output will look a bit different depending on your OS

- **ls** asks Jupyter to list files in the PWD (or `%ls`)
  - Note that `test.py` is there (on our computer, because we saved it there earlier)

- **cat test.py** asks Jupyter to print the contents of `test.py` (or `!type test.py` on Windows)
- **run test.py** runs the file and prints any output
But file X isn't in my PWD!

If you're trying to run a file not in the present working director, you'll get an error.

To fix this error you need to either

1. Shift the file into the PWD, or
2. Change the PWD to where the file lives

One way to achieve the first option is to use the Upload button

- The button is on the top level dashboard, where Jupyter first opened to
- Look where the pointer is in this picture

The second option can be achieved using the `cd` command

- On Windows it might look like this `cd C:/Python27/Scripts/dir`
- On Linux / OSX it might look like this `cd /home/user/scripts/dir`

Note: You can type the first letter or two of each directory name and then use the tab key to expand.

Loading Files

It's often convenient to be able to see your code before you run it.
In the following example we execute `load white_noise_plot.py` where `white_noise_plot.py` is in the PWD

(Use `%load` if automagic is off)

Now the code from the file appears in a cell ready to execute

![Jupyter Notebook screenshot](image)

**Saving Files**

To save the contents of a cell as file `foo.py`

- put `%file foo.py` as the first line of the cell
- Shift+Enter
Here %file is an example of a cell magic

1.2.7 Editors and IDEs

The preceding discussion covers most of what you need to know to interact with this website. However, as you start to write longer programs, you might want to experiment with your workflow. There are many different options and we mention them only in passing.

**JupyterLab**

JupyterLab is an integrated development environment centered around Jupyter notebooks. It is available through Anaconda and will soon be made the default environment for Jupyter notebooks. Reading the docs or searching for a recent YouTube video will give you more information.

**Text Editors**

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.

Among the most popular are Sublime Text and Atom. For a top quality open source text editor with a steeper learning curve, try Emacs. 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.

**Text Editors Plus IPython Shell**

A text editor is for writing programs. To run them you can continue to use Jupyter as described above. Another option is to use the excellent IPython shell. To use an IPython shell, open up a terminal and type ipython. You should see something like this.
The IPython shell has many of the features of the notebook: tab completion, color syntax, etc.

It also has command history through the arrow key

The up arrow key to brings previously typed commands to the prompt

This saves a lot of typing

Here’s one set up, on a Linux box, with

- a file being edited in Vim

- An IPython shell next to it, to run the file
IDEs

IDEs are Integrated Development Environments, which allow you to edit, execute and interact with code from an integrated environment.

One of the most popular in recent times is VS Code, which is now available via Anaconda.

We hear good things about VS Code; please tell us about your experiences on the forum.

### 1.2.8 Exercises

#### 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.

---

# 1.2. Setting up Your Python Environment
Now

1. Start your browser or open a new tab if its 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

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 well 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 an exercise, try

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

For example, if youve 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)

Even better,

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

### 1.3 An Introductory Example

#### Contents

- An Introductory Example
  - Overview
  - The Task: Plotting a White Noise Process
  - Version 1
  - Alternative Versions
  - Exercises
  - Solutions

We are now ready to start learning the Python language itself.

The level of this and the next few lectures will suit those with some basic knowledge of programming.

But don’t give up if you haven’t any — you are not excluded.

You just need to cover a few of the fundamentals of programming before returning here.

Good references for first-time programmers include:

- The first 5 or 6 chapters of How to Think Like a Computer Scientist
- Automate the Boring Stuff with Python
- The start of Dive into Python 3

Note: These references offer help on installing Python but you should probably stick with the method on our set up page.

You’ll then have an outstanding scientific computing environment (Anaconda) and be ready to move on to the rest of our course.

#### 1.3.1 Overview

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.

**Prerequisites**

The lecture on getting started with Python

### 1.3.2 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 $\epsilon_t$ is independent standard normal.

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

![White Noise Process Plot](image)

Well do this several different ways.
1.3.3 Version 1

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

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

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

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

**Import Statements**

The first two lines of the program import functionality

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*.

Here’s another example

```python
import numpy as np
np.sqrt(4)
```

```
2.0
```

We could also just write

```python
import numpy
numpy.sqrt(4)
```

```
2.0
```

But the former method is convenient and more standard

**Why all the imports?**

Remember that Python is a general purpose language

The core language is quite small so its easy to learn and maintain

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

Scientific work in Python is no exception

Most of our programs start off with lines similar to the *import* statements seen above

**Packages**

As stated above, NumPy is a Python *package*

Packages are used by developers to organize a code library

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 its located in

```
anaconda3/lib/python3.6/site-packages/numpy
```

### Subpackages

Consider the line $x = np.random.randn(100)$

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

You can see the contents [here](#).

Subpackages are just packages that are subdirectories of another package

### Importing Names Directly

Recall this code that we saw above

```
import numpy as np
np.sqrt(4)
```

```
2.0
```

Here is another way to access NumPys square root function

```
from numpy import sqrt
sqrt(4)
```

```
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 its harder for readers to know where `sqrt` came from, should they wish to

### 1.3.4 Alternative Versions

Lets try writing some alternative versions of our first program

Our aim in doing this is to illustrate some more Python syntax and semantics

The programs below are less efficient but

- help us understand basic constructs like loops
- illustrate common data types like lists

**A Version with a For Loop**

Here's a version that illustrates loops and Python lists

```python
ts_length = 100
_values = []  # Empty list

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

plt.plot(_values)
plt.show()
```

In brief,
- The first pair of lines `import` functionality as before
- The next line sets the desired length of the time series
• The next line creates an empty list called _values that will store the \( \epsilon_t \) values as we generate them
• The next three lines are the for loop, which repeatedly draws a new random number \( \epsilon_t \) and appends it to the end of the list _values
• The last two lines generate the plot and display it to the user

Lets study some parts of this program in more detail

Lists

Consider the statement _values = [], which creates an empty list

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

For example, try

```python
x = [10, 'foo', False]  # We can include heterogeneous data inside a list
type(x)
```

```
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)

```python
x
```

```
[10, 'foo', False]
```

```python
x.append(2.5)
x
```

```
[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 \( x \)

Well 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()
The full set of list methods can be found [here](#).

Following C, C++, Java, etc., lists in Python are zero based

```python
[x[0] for x in [10, 'foo', False]]
```

The For Loop

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

```python
for i in range(ts_length):
    e = np.random.randn()
    _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 `_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

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

If you put this in a text file or Jupyter cell and run it you will see

```
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

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

The Python interpreter performs the following:

- For each element of `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 well see soon enough

### Code Blocks and 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

### Tabs vs Spaces

One small gotcha here is the mixing of tabs and spaces, which often leads to errors.

(Important: Within text files, the internal representation of tabs and spaces is not the same)

You can use your `Tab` key to insert 4 spaces, but you need to make sure its configured to do so.

If you are using a Jupyter notebook you will have no problems here.

Also, good text editors will allow you to configure the Tab key to insert spaces instead of tabs trying searching on line

---

### 1.3. An Introductory Example

45
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:

```python
ts_length = 100
_values = []
i = 0
while i < ts_length:
    e = np.random.randn()
    _values.append(e)
    i = i + 1
plt.plot(_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`
User-Defined Functions

Now let's go back to the for loop, but restructure our program to make the logic clearer.

To this end, we will break our program into two parts:

1. A *user-defined function* that generates a list of random variables

2. The main part of the program that
   (a) calls this function to get data
   (b) plots the data

This is accomplished in the next program:

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

data = generate_data(100)
plt.plot(data)
plt.show()
```
Let’s go over this carefully, in case you’re not familiar with functions and how they work.

We have defined a function called `generate_data()` as follows:

- `def` is a Python keyword used to start function definitions.
- `def generate_data(n):` indicates that the function is called `generate_data`, and that it has a single argument `n`.
- The indented code is a code block called the *function body*; in this case, it creates an iid list of random draws using the same logic as before.
- The `return` keyword indicates that `_values` 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.

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 `_values` returned by the function.
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 by the next piece of code

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

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

![Plot of generated data](image)
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:

```python
def generate_data(n, generator_type):
    _values = []
    for i in range(n):
        e = generator_type()
        _values.append(e)
    return _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

```python
max(7, 2, 4)  # max() is a built-in Python function
```

```python
m = max
m(7, 2, 4)
```
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.

**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:

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

['dogs', 'cats', 'birds']

Here's another example:

```python
range(8)
```

[0, 1, 2, 3, 4, 5, 6, 7]

```python
doubles = [2 * x for x in range(8)]
doubles
```

[0, 2, 4, 6, 8, 10, 12, 14]

With the list comprehension syntax, we can simplify the lines:

```python
_values = []
for i in range(n):
    e = generator_type()
_values.append(e)
```

into

```python
_values = [generator_type() for i in range(n)]
```

**1.3.5 Exercises**

**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 \( n \).

**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 \( p \).

**Exercise 3**

Compute an approximation to \( \pi \) using Monte Carlo. Use no imports besides

```python
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 fall in \( B \) converges to the probability of landing in \( B \).
- For a circle, area = \( \pi \times \text{radius}^2 \).

**Exercise 4**

Write a program that prints one realization of the following random device:

- Flip an unbiased coin 10 times
- If 3 consecutive heads occur one or more times within this sequence, pay one dollar
- If not, pay nothing

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

**Exercise 5**

Your next 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
import numpy as np
import matplotlib.pyplot as plt

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

**Exercise 6**

To do the next exercise, you will need to know how to produce a plot legend

The following example should be sufficient to convey the idea

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

x = [np.random.randn() for i in range(100)]
plt.plot(x, label="white noise")
plt.legend()
plt.show()
```

Running it produces a figure like so

![Plot example](image)

Now, starting with your solution to exercise 5, plot three simulated time series, one for each of the cases $\alpha = 0$, $\alpha = 0.8$ and $\alpha = 0.98$

In particular, you should produce (modulo randomness) a figure that looks as follows
(The figure nicely illustrates how time series with the same one-step-ahead conditional volatilities, as these three processes have, can have very different unconditional volatilities.)

Use a for loop to step through the $\alpha$ values

Important 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
  - And if you omit the argument `'b-'` to the plot function, Matplotlib will automatically select different colors for each line
- The expression `'foo' + str(42)` evaluates to `'foo42'`

1.3.6 Solutions

Exercise 1

```python
def factorial(n):
    k = 1
    for i in range(n):
        k = k * (i + 1)
    return k
```
factorial(4)

24

Exercise 2

```python
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)
```

6

Exercise 3

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 $\pi$ then we can compute $A$ via $A = \pi r^2$

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

Summary: If we can estimate the area of the unit circle, then dividing by $r^2 = (1/2)^2 = 1/4$ gives an estimate of $\pi$

We estimate the area by sampling bivariate uniforms and looking at the fraction that fall into the unit circle

```python
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
```
Exercise 4

```python
from numpy.random import uniform

payoff = 0
count = 0

for i in range(10):
    U = uniform()
    count = count + 1 if U < 0.5 else 0
    if count == 3:
        payoff = 1

print(payoff)
```

1

Exercise 5

The next line embeds all subsequent figures in the browser itself

```python
alpha = 0.9
ts_length = 200
current_x = 0

x_values = []
for i in range(ts_length + 1):
    x_values.append(current_x)
    current_x = alpha * current_x + np.random.randn()
plt.plot(x_values)
plt.show()
```
Exercise 6

```python
as_ = [0.0, 0.8, 0.98]
ts_length = 200

for α in as_:
    x_values = []
    current_x = 0
    for i in range(ts_length):
        x_values.append(current_x)
        current_x = α * current_x + np.random.randn()
    plt.plot(x_values, label=f'α = {α}')
plt.legend()
plt.show()
```
1.4 Python Essentials

Contents

- Python Essentials
  - Data Types
  - Input and Output
  - Iterating
  - Comparisons and Logical Operators
  - More Functions
  - Coding Style and PEP8
  - Exercises
  - Solutions
In this lecture well cover features of the language that are essential to reading and writing Python code

1.4.1 Data Types

Weve already met several built in Python data types, such as strings, integers, floats and lists
Lets learn a bit more about them

**Primitive Data Types**

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

```python
x = True
x
```

True

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

```python
y = 100 < 10
y
```

False

```python
type(y)
```

`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

```python
x + y
```

1

```python
x * y
```

0

```python
True + True
```

2
bools = [True, True, False, True]  # List of Boolean values
sum(bools)

3

The two most common data types used to represent numbers are integers and floats

a, b = 1, 2
c, d = 2.5, 10.0
type(a)

```
int
```

type(c)

```
float
```

Computers distinguish between the two because, while floats are more informative, arithmetic operations on integers are faster and more accurate.

As long as you’re using Python 3.x, division of integers yields floats

```
1 / 2
```

```
0.5
```

But be careful! If you’re still using Python 2.x, division of two integers returns only the integer part.

For integer division in Python 3.x use this syntax:

```
1 // 2
```

```
0
```

Complex numbers are another primitive data type in Python

```
x = complex(1, 2)
y = complex(2, 1)
x + y
```

```
5j
```

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

```python
x = ('a', 'b')  # Parentheses instead of the square brackets
x = 'a', 'b'    # Or no brackets --- the meaning is identical
x
```

('a', 'b')

```python
type(x)
```

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

```python
x = [1, 2]
x[0] = 10
x
```

[10, 2]

But tuples are not

```python
x = (1, 2)
x[0] = 10
```

```
---------------------------------------------------------------------------
TypeError          Traceback (most recent call last)
<python-input-21-6cb4d74ca096> in <module>()
----> 1 x[0]=10

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

Well say more about the role of mutable and immutable data a bit later.

Tuples (and lists) can be unpacked as follows

```python
integers = (10, 20, 30)
x, y, z = integers
x
```

10

```python
y
```

20

Youve actually *seen an example of this* already
Tuple unpacking is convenient and well use it often

**Slice Notation**

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

For example,

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

```output
[4, 6, 8]
```

```python
a[1:3]
```

```output
[4, 6]
```

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

Negative numbers are also permissible

```python
a[-2:]
```

```output
[6, 8]
```

The same slice notation works on tuples and strings

```python
s = 'foobar'
s[-3:]
```

```output
'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

```python
d = {'name': 'Frodo', 'age': 33}
type(d)
```

```output
dict
```

```python
d['age']
```

```output
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

```python
def s1 = {'a', 'b'}
type(s1) # set

s2 = {'b', 'c'}
s1.issubset(s2)
False

s1.intersection(s2)
set(['b'])
```

The `set()` function creates sets from sequences

```python
s3 = set(('foo', 'bar', 'foo'))
s3
set({'foo', 'bar'})  # Unique elements only
```

### 1.4.2 Input and Output

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

```python
f = open('newfile.txt', 'w')  # Open 'newfile.txt' for writing
f.write('Testing
')  # Here '
' 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

```bash
%pwd
```

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

```python
f = open('newfile.txt', 'r')
out = f.read()
print(out)
```

```
'Testing
Testing again'
```

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

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

1.4.3 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.

Looping over Different Objects

Many Python objects are iterable, in the sense that they can 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.

```python
"""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
"""
```

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

1.4. Python Essentials
The program `us_cities.py` program reads the data in and makes the conversion:

```python
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()
```

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

The output is as follows:

<table>
<thead>
<tr>
<th>City</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>New York</td>
<td>8,244,910</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>3,819,702</td>
</tr>
<tr>
<td>Chicago</td>
<td>2,707,120</td>
</tr>
<tr>
<td>Houston</td>
<td>2,145,146</td>
</tr>
<tr>
<td>Philadelphia</td>
<td>1,536,471</td>
</tr>
<tr>
<td>Phoenix</td>
<td>1,469,471</td>
</tr>
<tr>
<td>San Antonio</td>
<td>1,359,758</td>
</tr>
<tr>
<td>San Diego</td>
<td>1,326,179</td>
</tr>
<tr>
<td>Dallas</td>
<td>1,223,229</td>
</tr>
</tbody>
</table>

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 `f` 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 well discuss some of them later on.

**Looping without Indices**

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

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

is preferred to

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

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

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

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

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

```python
{'John': 'F', 'Tom': 'E'}
```

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

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

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

The output of the loop is

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

### 1.4.4 Comparisons and Logical Operators

#### 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

```
x, y = 1, 2
x < y
```

```
True
```

```
x > y
```

```
False
```

One of the nice features of Python is that we can chain inequalities
As we saw earlier, when testing for equality we use `==`

\[
x = 1 \quad \# \text{Assignment}
x == 2 \quad \# \text{Comparison}
\]

False

For not equal use `!=`

\[
1 != 2
\]

True

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

\[
x = \text{'yes'} \text{ if } 42 \text{ else } \text{'no'}
x
\]

'yes'

\[
x = \text{'yes'} \text{ if } [] \text{ else } \text{'no'}
x
\]

'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

**Combining Expressions**

We can combine expressions using `and`, `or` and `not`
These are the standard logical connectives (conjunction, disjunction and denial)

\[
1 < 2 \text{ and 'f' in 'foo'}
\]

True

\[
1 < 2 \text{ and 'g' in 'foo'}
\]

False

\[
1 < 2 \text{ or 'g' in 'foo'}
\]

True

\[
\text{not True}
\]

False

\[
\text{not not True}
\]

True

Remember

- \( P \text{ and } Q \text{ is True if both are True, else False} \)
- \( P \text{ or } Q \text{ is False if both are False, else True} \)

### 1.4.5 More Functions

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

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

We have already met some

\[
\text{max}(19, 20)
\]

20

\[
\text{range}(4) \quad \# \text{ in python3 this returns a range iterator object}
\]

\[
\text{range}(0, 4)
\]

\[
\text{list(range}(4)) \quad \# \text{ will evaluate the range iterator and create a list}
\]
Two more useful built-in functions are \texttt{any()} and \texttt{all()}

\begin{verbatim}
bools = \texttt{False, True, True}
all(bools)  \# True if all are True and False otherwise
\end{verbatim}

\begin{verbatim}
False
\end{verbatim}

\begin{verbatim}
any(bools)  \# False if all are False and True otherwise
\end{verbatim}

\begin{verbatim}
True
\end{verbatim}

The full list of Python built-ins is \url{here}

Now lets talk some more about user-defined functions constructed using the keyword \texttt{def}

\section*{Why Write Functions?}

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

\begin{itemize}
  \item separating different strands of logic
  \item facilitating code reuse
\end{itemize}

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

The basics of user defined functions were discussed \url{here}

\section*{The Flexibility of Python Functions}

As we discussed in the \textit{previous lecture}, Python functions are very flexible

In particular

\begin{itemize}
  \item Any number of functions can be defined in a given file
  \item Functions can be (and often are) defined inside other functions
  \item Any object can be passed to a function as an argument, including other functions
  \item A function can return any kind of object, including functions
\end{itemize}
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:

```python
def f(x):
    if x < 0:
        return 'negative'
    return 'nonnegative'
```

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

### 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

```python
def f(x):
    ""
    This function squares its argument
    ""
    return x**2
```

After running this code, the docstring is available:

```python
f?
```

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

```python
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.
One-Line Functions: lambda

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

For example, the definitions

```python
def f(x):
    return x**3
```

and

```python
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:

```python
from scipy.integrate import quad
quad(lambda x: x**3, 0, 2)
```

(4.0, 4.440892098500626e-14)

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

Keyword Arguments

If you did the exercises in the previous lecture, you would have come across the statement

```python
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.
```python
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

\[
f(2)
\]

3

They can be modified as follows

\[
f(2, a=4, b=5)
\]

14

### 1.4.6 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 deviate from PEP8 in these lectures to better match mathematical notation)

### 1.4.7 Exercises

Solve the following exercises

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

**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
Exercise 2

Consider the polynomial

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

(1.1)

Write a function `p` such that `p(x, coeff)` that computes the value in (1.1) given a point `x` and a list of coefficients `coeff`

Try to use `enumerate()` in your loop

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'

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

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] < \ldots < point[n-1] = b`

Aim for clarity, not efficiency
1.4.8 Solutions

Exercise 1

Part 1 solution:

Here's one possible solution

```python
def sum(x * y for x, y in zip(x_vals, y_vals))
```

This also works

```python
sum(x * y for x, y in zip(x_vals, y_vals))
```

Part 2 solution:

One solution is

```python
sum([x % 2 == 0 for x in range(100)])
```

This also works:

```python
sum(x % 2 == 0 for x in range(100))
```

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

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

This is equivalent to

```python
sum([1 for x in range(100) if x % 2 == 0])
```
Part 3 solution

Here is one possibility:

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

2

Exercise 2

```python
def p(x, coeff):
    return sum(a * x**i for i, a in enumerate(coeff))
```

```python
p(1, (2, 4))
```

6

Exercise 3

Here is one solution:

```python
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')
```

3

Exercise 4

Here is a solution:

```python
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
```

```python
# == test == #
```
Of course if we use the `set` data type then the solution is easier

```python
def f(seq_a, seq_b):
    return set(seq_a).issubset(set(seq_b))
```

**Exercise 5**

```python
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)
```
1.5 OOP I: Introduction to Object Oriented Programming

Contents

• OOP I: Introduction to Object Oriented Programming
  – Overview
  – Objects
  – Summary

1.5.1 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

(Function in this context are referred to as methods)

Python and OOP

Python is 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

1.5.2 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

**Type**

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

For example

```python
s = 'This is a string'
print(type(s))

str
```

```python
x = 42  # Now let's create an integer
print(type(x))

int
```

The type of an object matters for many expressions

For example, the addition operator between two strings means concatenation

```python
'300' + 'cc'

'300cc'
```

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

```python
300 + 400

700
```

Consider the following expression

```python
'300' + 400
```

Here we are mixing types, and its 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,

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

```
700
```

### 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

```python
y = 2.5
z = 2.5
id(y)
```

```
166719660
```

```python
id(z)
```

```
166719740
```

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

### 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

```python
x = 42
x
```

```
42
```

```python
x.imag
```

```
```
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.

**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).

```python
x = ['foo', 'bar']
callable(x.append)
```

```
True
```

```python
callable(x.__doc__)
```

```
False
```

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

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

```
'THIS IS A STRING'
```

```python
s.lower()
```

```
'this is a string'
```
A great deal of Python functionality is organized around method calls

For example, consider the following piece of code

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

```
['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

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

```
['aa', 'b']
```

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

### 1.5.3 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

```python
def f(x): return x**2
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:

```python
f.__call__(3)
```

```
9
```

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

Modules loaded into memory are also treated as objects.

```python
import math
id(math)
```

```
3074329380L
```

This uniform treatment of data in Python (everything is an object) helps keep the language simple and consistent.
Next we cover the third party libraries most useful for scientific work in Python

2.1 NumPy

Contents

- NumPy
  - Overview
  - Introduction to NumPy
  - NumPy Arrays
  - Operations on Arrays
  - Additional Functionality
  - Exercises
  - Solutions

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

2.1.1 Overview

NumPy is a first-rate library for numerical programming

- Widely used in academia, finance and industry
- Mature, fast, stable and under continuous development

In this lecture we introduce NumPy arrays and the fundamental array processing operations provided by NumPy
References

- The official NumPy documentation

### 2.1.2 Introduction to NumPy

The essential problem that NumPy solves is fast array processing.

For example, suppose we want to create an array of 1 million random draws from a uniform distribution and compute the mean.

If we did this in pure Python it would be orders of magnitude slower than C or Fortran.

This is because

- Loops in Python over Python data types like lists carry significant overhead.
- C and Fortran code contains a lot of type information that can be used for optimization.
- Various optimizations can be carried out during compilation, when the compiler sees the instructions as a whole.

However, for a task like the one described above there is no need to switch back to C or Fortran.

Instead we can use NumPy, where the instructions look like this:

```python
import numpy as np

x = np.random.uniform(0, 1, size=1000000)
mean_x = x.mean()
```

The operations of creating the array and computing its mean are both passed out to carefully optimized machine code compiled from C.

More generally, NumPy sends operations in batches to optimized C and Fortran code.

This is similar in spirit to Matlab, which provides an interface to fast Fortran routines.

### A Comment on Vectorization

NumPy is great for operations that are naturally vectorized.

Vectorized operations are precompiled routines that can be sent in batches, like

- matrix multiplication and other linear algebra routines
- generating a vector of random numbers
- applying a fixed transformation (e.g., sine or cosine) to an entire array

In a later lecture we will discuss code that isn’t easy to vectorize and how such routines can also be optimized.
2.1.3 NumPy Arrays

The most important thing that NumPy defines is an array data type formally called `numpy.ndarray`.
NumPy arrays power a large proportion of the scientific Python ecosystem.

To create a NumPy array containing only zeros we use `np.zeros`:

```python
a = np.zeros(3)
a
```

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

```python
type(a)
```

```
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:

```python
a = np.zeros(3)
type(a[0])
```

```
numpy.float64
```

If we want to use integers we can specify as follows:

```python
a = np.zeros(3, dtype=int)
type(a[0])
```

```
numpy.int64
```

**Shape and Dimension**

Consider the following assignment:
```python
z = np.zeros(10)
```

Here \( z \) is a \textit{flat} array with no dimension neither row nor column vector

The dimension is recorded in the \texttt{shape} attribute, which is a tuple

```python
z.shape
```

\((10,)\)

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 \texttt{shape} attribute

```python
z.shape = (10, 1)
z
```

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

```python
z = np.zeros(4)
z.shape = (2, 2)
z
```

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

In the last case, to make the 2 by 2 array, we could also pass a tuple to the \texttt{zeros()} function, as in \( z = np.zeros((2, 2)) \)

### Creating Arrays

As we've seen, the \texttt{np.zeros} function creates an array of zeros

You can probably guess what \texttt{np.ones} creates

Related is \texttt{np.empty}, which creates arrays in memory that can later be populated with data

```python
z = np.empty(3)
z
```
array([0., 0., 1.])

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

\[ z = \text{np.linspace}(2, 4, 5) \quad \# \text{From 2 to 4, with 5 elements} \]

To create an identity matrix use either np.identity or np.eye

\[ z = \text{np.identity}(2) \]
\[ z \]

\[ \text{array([[1., 0.],} \]
\[ [0., 1.]]) \]

In addition, NumPy arrays can be created from Python lists, tuples, etc. using np.array

\[ z = \text{np.array}([[10, 20]]) \quad \# \text{ndarray from Python list} \]
\[ z \]

\[ \text{array([10, 20])} \]

\[ \text{type}(z) \]

\[ \text{numpy.ndarray} \]

\[ z = \text{np.array}([[10, 20], \text{dtype=]*float}]) \quad \# \text{Here 'float' is equivalent to 'np.float64'} \]
\[ z \]

\[ \text{array([[10., 20.}}} \]

\[ z = \text{np.array}([[1, 2], [3, 4]]) \quad \# \text{2D array from a list of lists} \]
\[ z \]

\[ \text{array([[1, 2],} \]
\[ [3, 4]]) \]

See also np.asarray, which performs a similar function, but does not make a distinct copy of data already in a NumPy array

\[ \text{na = np.linspace}(10, 20, 2) \]
\[ \text{na is np.asarray}(\text{na}) \quad \# \text{Does not copy NumPy arrays} \]
To read in the array data from a text file containing numeric data use \texttt{np.loadtxt} or \texttt{np.genfromtxt}. See the documentation for details.

**Array Indexing**

For a flat array, indexing is the same as Python sequences:

```python
z = np.linspace(1, 2, 5)
z
```

```
array([1. , 1.25, 1.5 , 1.75, 2. ])
```

```
z[0]
```

```
1.0
```

```python
z[0:2]  # Two elements, starting at element 0
```

```
array([1. , 1.25])
```

```
z[-1]
```

```
2.0
```

For 2D arrays the index syntax is as follows:

```python
z = np.array([[1, 2], [3, 4]])
z
```

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

```
z[0, 0]
```

```
1
```

```
z[0, 1]
```

```
```
And so on

Note that indices are still zero-based, to maintain compatibility with Python sequences

Columns and rows can be extracted as follows

\[ z[[0, :]] \]

\[
\begin{array}{c}
0 \quad 2 \\
\end{array}
\]

\[ z[:, 1] \]

\[
\begin{array}{c}
2 \quad 4 \\
\end{array}
\]

NumPy arrays of integers can also be used to extract elements

\[
z = np.linspace(2, 4, 5)
\]

\[
array([2. , 2.5, 3. , 3.5, 4. ])
\]

\[
indices = np.array((0, 2, 3))
\]

\[ z[indices] \]

\[
array([2. , 3. , 3.5])
\]

Finally, an array of dtype bool can be used to extract elements

\[
z[0, :]
\]

\[
array([1, 2])
\]

\[ z[:, 1] \]

\[
array([2, 4])
\]

\[
z = np.linspace(2, 4, 5)
\]

\[
array([2. , 2.5, 3. , 3.5, 4. ])
\]

\[
indices = np.array((0, 2, 3))
\]

\[ z[indices] \]

\[
array([2. , 3. , 3.5])
\]

\[
d = np.array([0, 1, 1, 0, 0], dtype=bool)
\]

\[
darray([False, True, True, False, False])
\]

\[ z[d] \]

\[
array([2.5, 3. ])
\]

Well see why this is useful below

An aside: all elements of an array can be set equal to one number using slice notation
```python
z = np.empty(3)
z
array([2., 3., 3.5])

z[:] = 42
z
array([42., 42., 42.])
```

**Array Methods**

Arrays have useful methods, all of which are carefully optimized

```python
a = np.array((4, 3, 2, 1))
a
array([4, 3, 2, 1])

a.sort()  # Sorts a in place
a
array([1, 2, 3, 4])

a.sum()  # Sum
10

a.mean()  # Mean
2.5

a.max()  # Max
4

a.argmax()  # Returns the index of the maximal element
3

a.cumsum()  # Cumulative sum of the elements of a
array([ 1, 3, 6, 10])
```
Another method worth knowing is `searchsorted()`

If \( z \) is a nondecreasing array, then \( z\text{.searchsorted}(a) \) returns the index of the first element of \( z \) that is \( \geq a \)

\[
z = \text{np.linspace}(2, 4, 5)
z
\]

\[
array([2. , 2.5, 3. , 3.5, 4. ])
\]

\[
z\text{.searchsorted}(2.2)
\]

\[
1
\]

Many of the methods discussed above have equivalent functions in the NumPy namespace

\[
a = \text{np.array}([4, 3, 2, 1])
a
\]

\[
\text{np.sum}(a)
\]

\[
10
\]

\[
\text{np.mean}(a)
\]

\[
2.5
\]
2.1.4 Operations on Arrays

Arithmetic Operations

The operators +, −, *, / and ** all act *elementwise* on arrays.

```python
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
a + b
```

```
array([ 6, 8, 10, 12])
```

```python
a * b
```

```
array([ 5, 12, 21, 32])
```

We can add a scalar to each element as follows:

```python
a + 10
```

```
array([11, 12, 13, 14])
```

Scalar multiplication is similar:

```python
a * 10
```

```
array([10, 20, 30, 40])
```

The two dimensional arrays follow the same general rules:

```python
A = np.ones((2, 2))
B = np.ones((2, 2))
A + B
```

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

```python
A + 10
```

```
array([[11., 11.],
       [11., 11.]])
```

```python
A * B
```

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

In particular, $A \times B$ is *not* the matrix product, it is an element-wise product.
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:

```python
A = np.ones((2, 2))
B = np.ones((2, 2))
A @ B
```

```
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

```python
A = np.array((1, 2))
B = np.array((10, 20))
A @ B
```

```
50
```

In fact, we can use @ when one element is a Python list or tuple

```python
A = np.array(((1, 2), (3, 4)))
A
```

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

```python
A @ (0, 1)
```

```
array([2, 4])
```

Since we are postmultiplying, the tuple is treated as a column vector

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:

```python
a = np.array([42, 44])
a
```

```
array([42, 44])
```
array([42, 44])

```
a[-1] = 0  # Change last element to 0
```

array([42, 0])

Mutability leads to the following behavior (which can be shocking to MATLAB programmers)

```
a = np.random.randn(3)
a
```

array([ 0.69695818, -0.05165053, -1.12617761])

```
b = a
b[0] = 0.0
```

array([ 0.0, -0.05165053, -1.12617761])

What 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`.

```
a = np.random.randn(3)
a
```

array([ 0.67357176, -0.16532174, 0.36539759])

```
b = np.copy(a)
b
```

array([ 0.67357176, -0.16532174, 0.36539759])
Now $b$ is an independent copy (called a *deep copy*)

```
b[:] = 1
b
```

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

```
a
```

```
array([0.67357176, -0.16532174, 0.36539759])
```

Note that the change to $b$ has not affected $a$

### 2.1.5 Additional Functionality

Let’s look at some other useful things we can do with NumPy

**Vectorized Functions**

NumPy provides versions of the standard functions $\log, \exp, \sin$, etc. that act *element-wise* on arrays

```
z = np.array([1, 2, 3])
np.sin(z)
```

```
array([ 0.84147098, 0.90929743, 0.14112001])
```

This eliminates the need for explicit element-by-element loops such as

```
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

```
z
```

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

```
(1 / np.sqrt(2 * np.pi)) * np.exp(-0.5 * z**2)
```

```
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 \texttt{ValueError}

```python
def f(x):
    return 1 if x > 0 else 0
```

The NumPy function \texttt{np.where} provides a vectorized alternative:

```python
x = np.random.randn(4)
x

array([-0.25521782, 0.38285891, -0.98037787, -0.083662])

np.where(x > 0, 1, 0)  # Insert 1 if x > 0 true, otherwise 0

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

You can also use \texttt{np.vectorize} to vectorize a given function

```python
def f(x):
    return 1 if x > 0 else 0
f = np.vectorize(f)
f(x)  # Passing the same vector x as in the previous example

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

However, this approach doesn't always obtain the same speed as a more carefully crafted vectorized function

**Comparisons**

As a rule, comparisons on arrays are done element-wise

```python
z = np.array([2, 3])
y = np.array([2, 3])
z == y

array([ True,  True])

y[0] = 5
z == y

array([False,  True])

z != y

array([ True, False])
```
The situation is similar for $>, <, \geq$ and $\leq$

We can also do comparisons against scalars

```python
z = np.linspace(0, 10, 5)
z
```

```output
array([ 0. , 2.5, 5. , 7.5, 10. ])
```

```python
z > 3
```

```output
array([False, False, True, True, True])
```

This is particularly useful for *conditional extraction*

```python
b = z > 3
b
```

```output
array([False, False, True, True, True])
```

```python
z[b]
```

```output
array([ 5. , 7.5, 10. ])
```

Of course we can and frequently do perform this in one step

```python
z[z > 3]
```

```output
array([ 5. , 7.5, 10. ])
```

### Subpackages

NumPy provides some additional functionality related to scientific programming through its subpackages

We've already seen how we can generate random variables using `np.random`

```python
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()
```

```output
5.096
```

Another commonly used subpackage is `np.linalg`

```python
A = np.array([[1, 2], [3, 4]])
np.linalg.det(A)  # Compute the determinant
```

2.1. NumPy
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.

### 2.1.6 Exercises

**Exercise 1**

Consider the polynomial expression

\[
p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_N x^N = \sum_{n=0}^{N} a_n x^n
\] (2.1)

Earlier, you wrote a simple function \( p(x, \text{coeff}) \) to evaluate (2.1) 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()`.

**Exercise 2**

Let \( q \) be a NumPy array of length \( n \) with \( q \cdot \text{sum}() == 1 \).

Suppose that \( q \) represents a probability mass function.

We wish to generate a discrete random variable \( x \) such that \( P(x = i) = q_i \).

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 \( I_0, I_1, \ldots, I_{n-1} \) 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.
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

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

2.1.7 Solutions

import matplotlib.pyplot as plt

Exercise 1

This code does the job

def p(x, coef):
    X = np.empty(len(coef))
    X[0] = 1
    X[1:] = x
    y = np.cumprod(X)  # y = [1, x, x**2,...]
    return coef @ y

Lets test it

2.1. NumPy
coef = np.ones(3)
print(coef)
print(p(1, coef))
# For comparison
q = np.poly1d(coef)
print(q(1))

[1. 1. 1.]
3.0
3.0

Exercise 2

Here’s our first pass at a solution:

```python
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

```python
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

**Exercise 3**

An example solution is given below.

In essence we’ve just taken this code from QuantEcon and added in a plot method

```python
""
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)
```

2.1. NumPy
```python
def plot(self, a=None, b=None):
    
    Plot the ecdf on the interval \([a, b]\).
    
    Parameters
    ----------
    a : scalar(float), optional(default=None)
        Lower end point of the plot interval
    b : scalar(float), optional(default=None)
        Upper end point 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__)
    plt.plot(x_vals, f(x_vals))
    plt.show()
```

Here's an example of usage

```python
X = np.random.randn(1000)
F = ECDF(X)
F.plot()
```
2.2 Matplotlib

2.2.1 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.

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.

2.2.2 The APIs

The MATLAB-style API

Here's the kind of easy example you might find in introductory treatments.
import matplotlib.pyplot as plt
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.

**The Object Oriented API**

Here is the code corresponding to the preceding figure using the object-oriented API.

```python
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

**Tweaks**

Here we've changed the line to red and added a legend

```python
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')`

```python
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

```python
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()
```

The figure now looks as follows

![Figure](image1.png)

Controlling the ticks, adding titles and so on is also straightforward

```python
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()
```

Here's the figure
2.2.3 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

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

```python
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()
```
Multiple Subplots

Sometimes we want multiple subplots in one figure

Here’s an example that generates 6 histograms

```python
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()
```

The output looks as follows
3D Plots

Matplotlib does a nice job of 3D plots. Here is one example.
```python
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()
```

**2.2. Matplotlib**
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.

```python
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()
```

Here’s the figure it produces (note axes through the origin and the grid):
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

### 2.2.4 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

### 2.2.5 Exercises

**Exercise 1**

Plot the function

\[ f(x) = \cos(\pi \theta x) \exp(-x) \]

over the interval \([0, 5]\) for each \(\theta\) in \(\text{np.linspace}(0, 2, 10)\)

Place all the curves in the same figure

The output should look like this
2.2.6 Solutions

Exercise 1

Here's one solution

```python
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, np.cos(np.pi * θ * x) * np.exp(-x))
plt.show()
```

2.3 SciPy

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.

### 2.3.1 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 the SciPy initialization file.

```python
# Import numpy symbols to scipy name space
import numpy as _num
linalg = None
from numpy import *
from numpy.random import rand, randn
from numpy.fft import fft, ifft
from numpy.lib.scimath import *

__all__ = []
__all__ += _num.__all__
__all__ += ['randn', 'rand', 'fft', 'ifft']

del _num
# Remove the linalg imported from numpy so that the scipy.linalg package can be imported.
del linalg
__all__.remove('linalg')
```

However, it's more common and better practice to use NumPy functionality explicitly.

```python
import numpy as np
a = np.identity(3)
```

What is useful in SciPy is the functionality in its subpackages:

- `scipy.optimize`, `scipy.integrate`, `scipy.stats`, etc.

These subpackages and their attributes need to be imported separately.

```python
from scipy.integrate import quad
from scipy.optimize import brentq
# etc
```
Lets explore some of the major subpackages

### 2.3.2 Statistics

The `scipy.stats` subpackage supplies
- numerous random variable objects (densities, cumulative distributions, random sampling, etc.)
- some estimation procedures
- some statistical tests

#### Random Variables and Distributions

Recall that `numpy.random` provides functions for generating random variables

```python
np.random.beta(5, 5, size=3)
```

```python
array([ 0.6167565 , 0.67994589, 0.32346476])
```

This generates a draw from the distribution below when $a, b = 5, 5$

$$f(x; a, b) = \frac{x^{(a-1)}(1-x)^{(b-1)}}{\int_0^1 u^{(a-1)}u^{(b-1)}du} \quad (0 \leq x \leq 1)$$  \hspace{1cm} (2.2)$$

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

```python
from scipy.stats import beta
import matplotlib.pyplot as plt

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(figsize=(10, 6))
ax.hist(obs, bins=40, normed=True)
ax.plot(grid, q.pdf(grid), 'k-', linewidth=2)
plt.show()
```

The following plot is produced
In this code we created a so-called `rv_frozen` object, via the call `q = beta(5, 5)`.

The frozen part of the notation implies that `q` represents a particular distribution with a particular set of parameters.

Once we've done so, we can then generate random numbers, evaluate the density, etc., all from this fixed distribution:

```python
q.cdf(0.4)  # Cumulative distribution function
0.2665676800000002

q.pdf(0.4)  # Density function
2.0901888000000004

q.ppf(0.8)  # Quantile (inverse cdf) function
0.63391348346427079

q.mean()
0.5
```

The general syntax for creating these objects is

2.3. SciPy
identifier = scipy.stats.distribution_name(shape_parameters)

where distribution_name is one of the distribution names in scipy.stats

There are also two keyword arguments, loc and scale, which following our example above, are called as

identifier = scipy.stats.distribution_name(shape_parameters, loc=c, scale=d)

These transform the original random variable $X$ into $Y = c + dX$

The methods rvs, pdf, cdf, etc. are transformed accordingly.

Before finishing this section, we note that there is an alternative way of calling the methods described above.

For example, the previous code can be replaced by

```python
obs = beta.rvs(5, 5, size=2000)
grid = np.linspace(0.01, 0.99, 100)

fig, ax = plt.subplots()
ax.hist(obs, bins=40, normed=True)
ax.plot(grid, beta.pdf(grid, 5, 5), 'k-', linewidth=2)
plt.show()
```

**Other Goodies in scipy.stats**

There are a variety statistical functions in scipy.stats.

For example, scipy.stats.linregress implements simple linear regression.

```python
from scipy.stats import linregress

x = np.random.randn(200)
y = 2 + x + 0.1 + np.random.randn(200)
gr, int, r_val, p_val, std_err = linregress(x, y)
gr, int

(1.9962554379482236, 0.008172822032671799)
```

To see the full list, consult the documentation.

**2.3.3 Roots and Fixed Points**

A root 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

$$f(x) = \sin(4(x - 1/4)) + x + x^{20} - 1$$

with $x \in [0, 1]$ we get
The unique root is approximately 0.408

Let's consider some numerical techniques for finding roots

**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 its less than 50
  – If yes, B asks if its less than 25
  – If no, B asks if its less than 75
And so on
This is bisection
Heres a fairly simplistic implementation of the algorithm in Python
It works for all sufficiently well behaved increasing continuous functions with \( f(a) < 0 < f(b) \)

```python
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 root is between lower and middle ===#
        if f(middle) > 0:
            lower, upper = lower, middle
        # === if root is between middle and upper ===#
        else:
            lower, upper = middle, upper
    return 0.5 * (upper + lower)
```

In fact SciPy provides its own bisection function, which we now test using the function \( f \) defined in (2.3)

```python
from scipy.optimize import bisect
bisect(f, 0, 1)
```

0.40829350427936706

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
This is a double-edged sword:

• When the function is well-behaved, the Newton-Raphson method is faster than bisection
• When the function is less well-behaved, the Newton-Raphson might fail
Let's investigate this using the same function \( f \), first looking at potential instability

```python
from scipy.optimize import newton
newton(f, 0.2)  # Start the search at initial condition \( x = 0.2 \)
```

0.40829350427935679

```python
newton(f, 0.7)  # Start the search at \( x = 0.7 \) instead
```

0.70017000000002816

The second initial condition leads to failure of convergence

On the other hand, using IPython's `timeit` magic, we see that `newton` can be much faster

```python
%timeit bisect(f, 0, 1)
```

1000 loops, best of 3: 261 us per loop

```python
%timeit newton(f, 0.2)
```

10000 loops, best of 3: 60.2 us per loop

### Hybrid Methods

So far we have seen that the Newton-Raphson method is fast but not robust

This bisection algorithm is robust but relatively slow

This illustrates a general principle

- If you have specific knowledge about your function, you might be able to exploit it to generate efficiency
- If not, then the algorithm choice involves a trade-off between speed of convergence 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

```python
brentq(f, 0, 1)
```
Here the correct solution is found and the speed is almost the same as `newton`.

**Multivariate Root Finding**

Use `scipy.optimize.fsolve`, a wrapper for a hybrid method in MINPACK.

See the documentation for details.

**Fixed Points**

SciPy has a function for finding (scalar) fixed points too.

```python
from scipy.optimize import fixed_point
fixed_point(lambda x: x**2, 10.0)  # 10.0 is an initial guess
```

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 $g(x) := x - f(x)$.

**2.3.4 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`

```python
from scipy.optimize import fminbound
fminbound(lambda x: x**2, -1, 2)  # Search in [-1, 2]
```
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.

2.3.5 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`.

```python
from scipy.integrate import quad

integral, error = quad(lambda x: x**2, 0, 1)
integral
```

```
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.

2.3.6 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.
2.3.7 Exercises

Exercise 1

Previously we discussed the concept of *recursive function calls*

Write a recursive implementation of the bisection function described above, which we repeat here for convenience.

```python
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 root is between lower and middle === #
        if f(middle) > 0:
            lower, upper = lower, middle
        # === if root is between middle and upper === #
        else:
            lower, upper = middle, upper
    return 0.5 * (upper + lower)
```

Test it on the function \( f = \lambda x: \sin(4 \times (x - 0.25)) + x + x^{20} - 1 \) discussed above.

2.3.8 Solutions

Exercise 1

Here's a reasonable solution:

```python
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
            bisect(f, lower, middle)
```

else:
    # Implies root is between middle and upper
    bisect(f, middle, upper)

We can test it as follows

```python
def f(x):
    return 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 = 0.41015625
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

2.4 Numba

- **Numba**
  - Overview
  - Where are the Bottlenecks?
  - Vectorization
  - Numba

2.4.1 Overview

In our lecture on *NumPy* we learned one method to improve speed and efficiency in numerical work.

That method, called *vectorization*, involved sending array processing operations in batch to efficient low level code.

This clever idea dates back to *Matlab*, which uses it extensively.

Unfortunately, vectorization is limited and has several weaknesses.

One weakness is that it is highly memory intensive.
Another problem is that only some algorithms can be vectorized.

In the last few years, a new Python library called Numba has appeared that solves many of these problems.

It does so through something called just in time (JIT) compilation.

JIT compilation is effective in many numerical settings and can generate extremely fast, efficient code.

It can also do other tricks such as facilitate multithreading (a form of parallelization well suited to numerical work).

The Need for Speed

To understand what Numba does and why, we need some background knowledge.

Let's start by thinking about higher level languages, such as Python.

These languages are optimized for humans.

This means that the programmer can leave many details to the runtime environment, such as:

- 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.

High productivity languages should be chosen over high speed languages for the great majority of scientific computing tasks.

This is because:

1. Of any given program, relatively few lines are ever going to be time-critical.
2. For those lines of code that are time-critical, we can achieve C-like speed using a combination of NumPy and Numba.

This lecture provides a guide.

2.4.2 Where are the Bottlenecks?

Let's start by trying to understand why high level languages like Python are slower than compiled code.
Dynamic Typing

Consider this Python operation

```python
a, b = 10, 10
a + b
```

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.

```python
a, b = 'foo', 'bar'
a + b
```

'foobar'

If `a` and `b` are lists, then `a + b` requires list concatenation.

```python
a, b = ['foo'], ['bar']
a + b
```

['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.

```c
#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;
}
```

2.4. Numba
The variables \( i \) and \( \text{sum} \) are explicitly declared to be integers.
Hence, the meaning of addition here is completely unambiguous.

**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.
2.4.3 Vectorization

Vectorization is about sending batches of related operations to native machine code

- The machine code itself is typically compiled from carefully optimized C or Fortran

This can greatly accelerate many (but not all) numerical computations

Operations on Arrays

First let's run some imports

```python
import random
import numpy as np
import quantecon as qe
```

Now let's try this non-vectorized code

```python
qe.util.tic()   # Start timing
n = 100_000
sum = 0
for i in range(n):
    x = random.uniform(0, 1)
    sum += x**2
qe.util.toc()   # End timing
```

TOC: Elapsed: 0.055155277252197266 seconds.

Now compare this vectorized code

```python
qe.util.tic()
n = 100_000
x = np.random.uniform(0, 1, n)
np.sum(x**2)
qe.util.toc()
```

TOC: Elapsed: 0.001653194274902344 seconds.

The second code block, which achieves the same thing as the first, runs much faster

The reason is that in the second implementation we have broken 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

2.4. Numba
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

**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:

```python
np.cos(1.0)
```

```
0.54030230586813977
```

```python
np.cos(np.linspace(0, 1, 3))
```

```
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 $a$ lets choose

$$f(x, y) = \frac{\cos(x^2 + y^2)}{1 + x^2 + y^2} \quad \text{and} \quad a = 3$$

Heres a plot of $f$

```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d 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, 
```
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

Here's a non-vectorized version that uses Python loops

```python
def f(x, y):
    return np.cos(x**2 + y**2) / (1 + x**2 + y**2)
```

```
grid = np.linspace(-3, 3, 1000)
m = -np.inf
qe.tic()
for x in grid:
    for y in grid:
        z = f(x, y)
```

2.4. *Numba*
if \( z > m \):
  \( m = z \)

And heres a vectorized version

```python
def f(x, y):
    return np.cos(x**2 + y**2) / (1 + x**2 + y**2)

grid = np.linspace(-3, 3, 1000)
x, y = np.meshgrid(grid, grid)

qe.tic()
np.max(f(x, y))
qe.toc()
```

In the vectorized version, all the looping takes place in compiled code

As you can see, the second version is much faster

(Well make it even faster again below, when we discuss Numba)

**Pros and Cons of Vectorization**

At its best, vectorization yields fast, simple code

However, its 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

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 nice ways to speed up Python loops

**2.4.4 Numba**

One exciting development in this direction is Numba

Numba aims to automatically compile functions to native machine code instructions on the fly

The process isnt flawless, since Numba needs to infer type information on all variables to generate pure machine instructions
Such inference isn’t possible in every setting
But for simple routines Numba infers types very well
Moreover, the hot loops at the heart of our code that we need to speed up are often such simple routines

**Prerequisites**

If you followed our set up instructions, then Numba should be installed

Make sure you have the latest version of Anaconda by running `conda update anaconda` from a terminal (Mac, Linux) / Anaconda command prompt (Windows)

**An Example**

Let’s consider some problems that are difficult to vectorize

One is generating the trajectory of a difference equation given an initial condition

Let’s take the difference equation to be the quadratic map

\[ x_{t+1} = 4x_t(1 - x_t) \]

Here’s the plot of a typical trajectory, starting from \( x_0 = 0.1 \), with \( t \) on the x-axis

```python
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

x = qm(0.1, 250)
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(x, 'b-', lw=2, alpha=0.8)
ax.set_xlabel('time', fontsize=16)
plt.show()
```
To speed this up using Numba is trivial using Numbas jit function

```python
from numba import jit
qm_numba = jit(qm)  # qm_numba is now a 'compiled' version of qm
```

Lets time and compare identical function calls across these two versions:

```python
qe.util.tic()
qm(0.1, int(10**5))
time1 = qe.util.toc()

TOC: Elapsed: 0.07170653343200684 seconds.

qe.util.tic()
qm_numba(0.1, int(10**5))
time2 = qe.util.toc()

TOC: Elapsed: 0.06515693664550781 seconds.
```

The first execution is relatively slow because of JIT compilation (see below)

Next time and all subsequent times it runs much faster:

```python
qe.util.tic()
qm_numba(0.1, int(10**5))
```
time2 = qe.util.toc()

TOC: Elapsed: 0.0003921985626220703 seconds.

time1 / time2  # Calculate speed gain

182.8322188449848

That’s a speed increase of two orders of magnitude!

Your mileage will of course vary depending on hardware and so on

Nonetheless, two orders of magnitude is huge relative to how simple and clear the implementation is

Decorator Notation

If you don’t need a separate name for the numbaified version of qm, you can just put @jit before the function

```python
@jit
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
```

This is equivalent to qm = jit(qm)

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

As you can imagine, this is easier for simple Python objects (simple scalar data types, such as floats, integers, etc.)

Numba also plays well with NumPy arrays, which it treats as typed memory regions

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 some code is generated using the Python runtime

In this second setting, Numba typically provides only minor speed gains or none at all

Hence, its prudent when using Numba to focus on speeding up small, time-critical snippets of code

2.4. Numba
This will give you much better performance than blanketing your Python programs with @jit statements.

**A Gotcha: Global Variables**

Consider the following example:

```python
a = 1
@jit
def add_x(x):
    return a + x

print(add_x(10))

11

a = 2
print(add_x(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.

**Numba for vectorization**

Numba can also be used to create custom *ufuncs* with the @vectorize decorator.

To illustrate the advantage of using Numba to vectorize a function, we return to a maximization problem discussed above:

```python
from numba import vectorize

@vectorize
def f_vec(x, y):
    return np.cos(x**2 + y**2) / (1 + x**2 + y**2)

grid = np.linspace(-3, 3, 1000)
x, y = np.meshgrid(grid, grid)

np.max(f_vec(x, y))  # Run once to compile
qe.tic()
np.max(f_vec(x, y))
qe.toc()
```
This is faster than our vectorized version using NumPys ufuncs.

Why should that be? After all, anything vectorized with NumPy will be running in fast C or Fortran code. The reason is that its much less memory intensive.

For example, when NumPy computes `np.cos(x**2 + y**2)` it first creates the intermediate arrays `x**2` and `y**2`, then it creates the array `np.cos(x**2 + y**2)`.

In our `@vectorize` version using Numba, the entire operator is reduced to a single vectorized process and none of these intermediate arrays are created.

We can gain further speed improvements using Numbas automatic parallelization feature by specifying `target=parallel`.

In this case, we need to specify the types of our inputs and outputs.

```python
@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
qe.tic()
np.max(f_vec(x, y))
ze.toc()
```

This is a striking speed up with very little effort.

### 2.5 Other Scientific Libraries

**Contents**

- Other Scientific Libraries
  - Overview
  - Cython
  - Joblib
  - Other Options
  - Exercises
  - Solutions

2.5. Other Scientific Libraries 139
2.5.1 Overview

In this lecture we review some other scientific libraries that are useful for economic research and analysis. We have, however, already picked most of the low hanging fruit in terms of economic research. Hence you should feel free to skip this lecture on first pass.

2.5.2 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.

**Important Note:**

In what follows code is executed in a Jupyter notebook. This is to take advantage of a Cython cell magic that makes Cython particularly easy to use. Some modifications are required to run the code outside a notebook.

- See the book *Cython* by Kurt Smith or the online documentation.

A First Example

Let’s start with a rather artificial example. Suppose that we want to compute the sum \( \sum_{i=0}^{n} \alpha^i \) for given \( \alpha, n \). Suppose further that we’ve forgotten the basic formula

\[
\sum_{i=0}^{n} \alpha^i = \frac{1 - \alpha^{n+1}}{1 - \alpha}
\]

for a geometric progression and hence have resolved to rely on a loop.

Python vs C

Here’s a pure Python function that does the job.
def geo_prog(alpha, n):
    current = 1.0
    sum = current
    for i in range(n):
        current = current * alpha
        sum = sum + current
    return sum

This works fine but for large $n$ it is slow

Here is a C function that will do the same thing

double geo_prog(double alpha, int n) {
    double current = 1.0;
    double sum = current;
    int i;
    for (i = 1; i <= n; i++) {
        current = current * alpha;
        sum = sum + current;
    }
    return sum;
}

If you’re not familiar with C, the main thing you should take notice of is the type definitions

- int means integer
- double means double precision floating point number
- the double in double geo_prog(...) indicates that the function will return a double

Not surprisingly, the C code is faster than the Python code

**A Cython Implementation**

Cython implementations look like a convex combination of Python and C

We’re going to run our Cython code in the Jupyter notebook, so we’ll start by loading the Cython extension in a notebook cell

```
%load_ext Cython
```

In the next cell, we execute the following

```
%%cython
def geo_prog_cython(double alpha, int n):
    cdef double current = 1.0
    cdef double sum = current
    cdef int i
    for i in range(n):
        current = current * alpha
        sum = sum + current
    return sum
```
Here `cdef` is a Cython keyword indicating a variable declaration, and is followed by a type. The `%cython` line at the top is not actually Cython code; it's a Jupyter cell magic indicating the start of Cython code.

After executing the cell, you can now call the function `geo_prog_cython` from within Python.

What you are in fact calling is compiled C code with a Python call interface.

```python
import quantecon as qe
qe.util.tic()
geo_prog(0.99, int(10**6))
qe.util.toc()
```

TOC: Elapsed: 0.11026620864868164 seconds.

```python
qe.util.tic()
geo_prog_cython(0.99, int(10**6))
qe.util.toc()
```

TOC: Elapsed: 0.038515567779541016 seconds.

**Example 2: Cython with NumPy Arrays**

Let's go back to the first problem that we worked with: generating the iterates of the quadratic map

\[ x_{t+1} = 4x_t(1 - x_t) \]

The problem of computing iterates and returning a time series requires us to work with arrays. The natural array type to work with is NumPy arrays.

Here's a Cython implementation that initializes, populates and returns a NumPy array:

```python
%%cython
import numpy as np
def qm_cython_first_pass(double x0, int n):
    cdef int t
    x = np.zeros(n+1, float)
x[0] = x0
    for t in range(n):
        x[t+1] = 4.0 * x[t] * (1 - x[t])
    return np.asarray(x)
```

If you run this code and time it, you will see that its performance is disappointing—nothing like the speed gain we got from Numba.

```python
qe.util.tic()
qm_cython_first_pass(0.1, int(10**5))
qe.util.toc()
```
This example was also computed in the Numba lecture, and you can see Numba is around 90 times faster. The reason is that working with NumPy arrays incurs substantial Python overheads. We can do better by using Cythons typed memoryviews, which provide more direct access to arrays in memory.

When using them, the first step is to create a NumPy array. Next, we declare a memoryview and bind it to the NumPy array. Here’s an example:

```
@cython
import numpy as np
from numpy cimport float_t

def qm_cython(double x0, int n):
    cdef int t
    x_np_array = np.zeros(n+1, dtype=float)
    cdef float_t [:] x = x_np_array
    x[0] = x0
    for t in range(n):
        x[t+1] = 4.0 * x[t] * (1 - x[t])
    return np.asarray(x)
```

Here:

- `cimport` pulls in some compile-time information from NumPy
- `cdef float_t [:] x = x_np_array` creates a memoryview on the NumPy array `x_np_array`
- The return statement uses `np.asarray(x)` to convert the memoryview back to a NumPy array

Let’s time it:

```
qe.util.tic()
qm_cython(0.1, int(10**5))
qe.util.toc()
```

This is fast, although still slightly slower than `qm_numba`

**Summary**

Cython requires more expertise than Numba, and is a little more fiddly in terms of getting good performance. In fact, it’s surprising how difficult it is to beat the speed improvements provided by Numba.

Nonetheless,

2.5. Other Scientific Libraries
• Cython is a very mature, stable and widely used tool
• Cython can be more useful than Numba when working with larger, more sophisticated applications

2.5.3 Joblib

Joblib is a popular Python library for caching and parallelization

To install it, start Jupyter and type

```bash
!pip install joblib
```

from within a notebook

Here we review just the basics

Caching

Perhaps, like us, you sometimes run a long computation that simulates a model at a given set of parameters to generate a figure, say, or a table

20 minutes later you realize that you want to tweak the figure and now you have to do it all again

What caching will do is automatically store results at each parameterization

With Joblib, results are compressed and stored on file, and automatically served back up to you when you repeat the calculation

An Example

Lets look at a toy example, related to the quadratic map model discussed above

Lets say we want to generate a long trajectory from a certain initial condition $x_0$ and see what fraction of the sample is below 0.1

(Well omit JIT compilation or other speed ups for simplicity)

Heres our code

```python
from joblib import Memory

memory = Memory(cachedir='./joblib_cache')

@memory.cache
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 np.mean(x < 0.1)
```
We are using `joblib` to cache the result of calling `qm` at a given set of parameters

With the argument `cachedir=./joblib_cache`, any call to this function results in both the input values and output values being stored in a subdirectory `joblib_cache` of the present working directory

(In UNIX shells, `.` refers to the present working directory)

The first time we call the function with a given set of parameters we see some extra output that notes information being cached

```python
qe.util.tic()
n = int(1e7)
qm(0.2, n)
qe.util.toc()
```

```
[Memory] Calling __main__ [truncated]
qm  -  6.9s, 0.1min
```

The next time we call the function with the same set of parameters, the result is returned almost instantaneously

```python
qe.util.tic()
n = int(1e7)
qm(0.2, n)
qe.util.toc()
```

```
0.204758079524
TOC: Elapsed: 0.0009872913360595703 seconds.
```

### 2.5.4 Other Options

There are in fact many other approaches to speeding up your Python code

One is interfacing with Fortran

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 very 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

### 2.5.5 Exercises

**Exercise 1**

Later well 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 month, and suppose the current state is high. We see from the graph that the state next month will be:

- high with probability 0.8
- low with probability 0.2

Your task is to simulate a sequence of monthly volatility states according to this rule. Set the length of the sequence to \( n = 100000 \) and start in the high state. Implement a pure Python version, a Numba version and a Cython 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.

2.5.6 Solutions

Exercise 1

We let:

- 0 represent low
- 1 represent high

\[ p, q = 0.1, 0.2 \] # Prob of leaving low and high state respectively

Here’s a pure Python version of the function:

```python
def compute_series(n):
    x = np.empty(n, dtype=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:
```

Chapter 2. The Scientific Libraries
```python
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

```python
n = 100000
x = compute_series(n)
print(np.mean(x == 0))  # Fraction of time x is in state 0
```

0.66951

Now let's time it

```python
qe.util.tic()
compute_series(n)
qe.util.toc()
```

TOC: Elapsed: 0.07770729064941406 seconds.

Next let's implement a Numba version, which is easy

```python
from numba import jit
compute_series_numba = jit(compute_series)
```

Let's check we still get the right numbers

```python
x = compute_series_numba(n)
print(np.mean(x == 0))
```

0.66764

Let's see the time

```python
qe.util.tic()
compute_series_numba(n)
qe.util.toc()
```

TOC: Elapsed: 0.0017528533935546875 seconds.

This is a nice speed improvement for one line of code

Now let's implement a Cython version

```python
%load_ext Cython

%%cython
import numpy as np
```
from numpy cimport int_t, float_t

def compute_series_cy(int n):
    # == Create NumPy arrays first == #
    x_np = np.empty(n, dtype=int)
    U_np = np.random.uniform(0, 1, size=n)
    # == Now create memoryviews of the arrays == #
    cdef int_t [:] x = x_np
    cdef float_t [:] U = U_np
    # == Other variable declarations == #
    cdef float p = 0.1
    cdef float q = 0.2
    cdef int t
    # == Main loop == #
    x[0] = 1
    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 np.asarray(x)

compute_series_cy(10)

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

x = compute_series_cy(n)
print(np.mean(x == 0))

0.66927

qe.util.tic()
compute_series_cy(n)
qe.util.toc()

TOC: Elapsed: 0.0025839805603027344 seconds.

The Cython implementation is fast, but not as fast as Numba
This part provides a look at more advanced concepts in Python programming

### 3.1 Writing Good Code

#### Contents

- **Writing Good Code**
  - Overview
  - An Example of Bad Code
  - Good Coding Practice
  - Revisiting the Example
  - Summary

#### 3.1.1 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
### 3.1.2 An Example of Bad 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

\[
k_{t+1} = sk_t^\alpha + (1 - \delta)k_t, \quad t = 0, 1, 2, \ldots
\]  

(3.1)

Here

- \(k_t\) is capital at time \(t\) and
- \(s, \alpha, \delta\) are parameters (savings, a productivity parameter and depreciation)

For each parameterization, the code

1. sets \(k_0 = 1\)
2. iterates using (3.1) to produce a sequence \(k_0, k_1, k_2, \ldots, k_T\)
3. plots the sequence

The plots will be grouped into three subfigures.

In each subfigure, two parameters are held fixed while another varies.

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

# Allocate memory for time series
k = np.empty(50)

fig, axes = plt.subplots(3, 1, figsize=(12, 15))

# Trajectories with different \(\alpha\)
\(\delta = 0.1\)
\(s = 0.4\)
\(\alpha = (0.25, 0.33, 0.45)\)

for j in range(3):
    k[0] = 1
    for t in range(49):
        k[t+1] = s * k[t]**\alpha[j] + (1 - \delta) * k[t]
    axes[0].plot(k, 'o-', label=r'\$\alpha = \{\alpha[j]\}, \; s = \{s\}, \; \delta = \{\delta\}$

axes[0].grid(lw=0.2)
axes[0].set_yscale('log')
axes[0].set_xlabel('time')
axes[0].set_ylabel('capital')
axes[0].legend(loc='upper left', frameon=True, fontsize=14)

# Trajectories with different \(s\)
\(\delta = 0.1\)
```
\(\alpha = 0.33\)
\(s = (0.3, 0.4, 0.5)\)

```python
for j in range(3):
    k[0] = 1
    for t in range(49):
        k[t+1] = s[j] * k[t] + (1 - \delta) * k[t]
    axes[1].plot(k, 'o-', label=rf'\(\alpha = (\alpha), \ s = (s), \ \delta = (\delta)\)')
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, fontsize=14)

# Trajectories with different \(\delta\)
\(\delta = (0.05, 0.1, 0.15)\)
\(\alpha = 0.33\)
\(s = 0.4\)

for j in range(3):
    k[0] = 1
    for t in range(49):
        k[t+1] = s * k[t] + (1 - \delta[j]) * k[t]
    axes[2].plot(k, 'o-', label=rf'\(\alpha = (\alpha), \ s = (s), \ \delta = (\delta[j])\)')
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, fontsize=14)
```

plt.show()
True, the code more or less follows PEP8

At the same time, its very poorly structured

Lets talk about why thats the case, and what we can do about it
3.1.3 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:

**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 complement.

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

**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.

Well talk about how to avoid repetition below.
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 discuss this just below)

JIT Compilation

In fact, there’s now another good reason to avoid global variables. In scientific computing, we’re witnessing the rapid growth of just in time (JIT) compilation. JIT compilation can generate excellent performance for scripting languages like Python and Julia. But the task of the compiler used for JIT compilation becomes much harder when many global variables are present. (This is because data type instability hinders the generation of efficient machine code. We learn more about such topics later on)

Use Functions or Classes

Fortunately, we can easily avoid the evils of global variables and WET code.

- WET stands for we love 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 learn more about these topics over time. (Personal preference is part of the story too)
Whats really important is that you use one or the other or both

### 3.1.4 Revisiting the Example

Heres some code that reproduces the plot above with better coding style

It uses a function to avoid repetition

Note also that

- global variables are quarantined by collecting 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

```python
def plot_path(ax, α, s, s_vals, δs, series_length=50):
    
    Add a time series plot to the axes ax for all given parameters.
    
    k = np.empty(series_length)
    
    for (α, s, δ) in product(αs, s_vals, δs):
        k[0] = 1
        for t in range(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.grid(lw=0.2)
    ax.set_xlabel('time')
    ax.set_ylabel('capital')
    ax.set_ylim(0, 18)
    ax.legend(loc='upper left', frameon=True, fontsize=14)

fig, axes = plt.subplots(3, 1, figsize=(12, 15))

# 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()
```

---

### 3.1. Writing Good Code 155
3.1.5 Summary

Writing decent code isn’t hard

It’s also fun and intellectually satisfying
We recommend that you cultivate good habits and style even when you write relatively short programs

### 3.2 OOP II: Building Classes

#### Contents

- **OOP II: Building Classes**
  - Overview
  - OOP Review
  - Defining Your Own Classes
  - Special Methods
  - Exercises
  - Solutions

#### 3.2.1 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 will provide plenty of examples.
3.2.2 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

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.

```python
x = [1, 5, 4]
x.sort()
x
```

```
[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:

```python
x = [1, 5, 4]
x.sort()
x.__class__
```
list

- \( x \) is an object or instance, created from the definition for Python lists, but with its own particular data
- \( x.sort() \) and \( x._\texttt{class}__ \) are two attributes of \( x \)
- \( \texttt{dir}(x) \) can be used to view all the attributes of \( x \)

### Why is OOP Useful?

OOP is useful for the same reason that abstraction is useful: for recognizing and exploiting 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., \texttt{simulate})

When we use OOP, the \texttt{simulate} method is conveniently bundled together with the Markov chain object

### 3.2.3 Defining Your Own Classes

Let's build some simple classes to start off

**Example: A Consumer Class**

First, we'll build a Consumer class with

- a \texttt{wealth} attribute that stores the consumer's wealth (data)
- an \texttt{earn} method, where \texttt{earn(y)} increments the consumer's wealth by \( y \)
- a \texttt{spend} method, where \texttt{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
```python
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("Insufficient 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.

Well also discuss the role of `self` just below.

**Usage**

Here's an example of usage:

```python
c1 = Consumer(10)  # Create instance with initial wealth 10
c1.spend(5)
c1.wealth
```

```
5
```

```python
c1.earn(15)
c1.spend(100)
```

```
```
Insufficient funds

We can of course create multiple instances each with its own data

```python
c1 = Consumer(10)
c2 = Consumer(12)
c2.spend(4)
c2.wealth
```

8

```python
c1.wealth
```

10

In fact each instance stores its data in a separate namespace dictionary

```python
c1.__dict__
```

```python
{'wealth': 10}
```

```python
c2.__dict__
```

```python
{'wealth': 8}
```

When we access or set attributes were actually just modifying the dictionary maintained by the instance

**Self**

If you look at the Consumer class definition again youll 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:

```python
print(Consumer.__dict__)  # Show __dict__ attribute of class object
```

```python
{'earn': <function Consumer.earn at 0x7f2590054d90>,
'spend': <function Consumer.spend at 0x7f2590054e18>,
'__doc__': None,
'__weakref__': <attribute '__weakref__' of 'Consumer' objects>,
'__init__': <function Consumer.p__init__ at 0x7f2590054d08>,
'__module__': '__main__',
'__dict__': <attribute '__dict__' of 'Consumer' objects>}
```

Note how the three methods `__init__`, `earn` and `spend` are stored in the class object.

Consider the following code:

```python
c1 = Consumer(10)
c1.earn(10)
c1.wealth
```

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:

```python
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`.

**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:
Here

- $s$ is an exogenously given savings rate
- $z$ is a productivity parameter
- $\alpha$ is capital's share of income
- $n$ is the population growth rate
- $\delta$ is the depreciation rate

The **steady state** of the model is the $k$ that solves (3.2) 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 (3.2)
- The `update` method uses `h` to update capital as per (3.2)
  - 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

```python
class Solow:
    r"""
    Implements the Solow growth model with update rule

    $k_{t+1} = \frac{(szk_t^\alpha + (1 - \delta)k_t)}{1 + n}$
    """

    def __init__(self, n=0.05, # population growth rate
                 s=0.25, # savings rate
                 \delta=0.1, # depreciation rate
                 \alpha=0.3, # share of labor
                 z=2.0, # productivity
                 k=1.0): # current capital stock
        self.n, self.s, self.\delta, self.\alpha, self.z = n, s, \delta, \alpha, z
        self.k = k

    def h(self):
        """Evaluate the h function"
        # Unpack parameters (get rid of self to simplify notation)
        n, s, \delta, \alpha, z = self.n, self.s, self.\delta, self.\alpha, self.z
        # Apply the update rule
        return (s * z * self.k**\alpha + (1 - \delta) * self.k) / (1 + n)

    def update(self):
```

3.2. OOP II: Building Classes
"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, delta, alpha, z = self.n, self.s, self.delta, self.alpha, self.z
    # Compute and return steady state
    return (s*z / (n + delta))**(1 / (1 - alpha))

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

```python
import matplotlib.pyplot as plt

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

Here's the figure it produces
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:

```python
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:
```

3.2. OOP II: Building Classes

165
```python
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 is a sample of usage:

```python
baseline_params = 15, .5, -2, .5, 3
m = Market(*baseline_params)
print("equilibrium price = ", m.price())
equilibrium price = 18.5

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:

```python
import numpy as np

# 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 figure produced looks as follows:

![Graph of inverse demand and supply curves with and without tax](image)

The next program provides a function that

- takes an instance of `Market` as a parameter
Computes deadweight loss from the imposition of the tax

```python
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

```python
baseline_params = 15, .5, -2, .5, 3
m = Market(*baseline_params)
deadw(m)  # Show deadweight loss
```

1.125

**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

\[ x_{t+1} = r x_t (1 - x_t), \quad x_0 \in [0, 1], \quad r \in [0, 4] \]  

(3.3)

Let's write a class for generating time series from this model.

Here's one implementation

```python
class Chaos:
    """Models the dynamical system with \(x_{t+1} = r x_t (1 - x_t)\)""
    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)
```

168 Chapter 3. Advanced Python Programming
def update(self):
    return path

Here's an example of usage

```
ch = Chaos(0.1, 4.0)  # x0 = 0.1 and r = 0.4
ch.generate_sequence(5)  # First 5 iterates
```

```
[0.1, 0.36000000000000004, 0.9216, 0.28901376000000006, 0.8219392261226498]
```

This piece of code plots a longer trajectory

```
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 resulting figure looks as follows

![Bifurcation diagram](image)

The next piece of code provides a bifurcation diagram

```
fig, ax = plt.subplots()
ch = Chaos(0.1, 4)
```
Here is the figure it generates

On the horizontal axis is the parameter $r$ in (3.3)

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.

### 3.2.4 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.

```python
x = (10, 20)
len(x)
```

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.

```python
class Foo:
    def __len__(self):
        return 42
```

Now we get

```python
f = Foo()
len(f)
```

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.

```python
class Foo:
    def __call__(self, x):
        return x + 42
```

After running we get

```python
f = Foo()
f(8)  # Exactly equivalent to f.__call__(8)
```

50

Exercise 1 provides a more useful example.
3.2.5 Exercises

Exercise 1

The empirical cumulative distribution function (ecdf) corresponding to a sample \( \{X_i\}_{i=1}^n \) is defined as

\[
F_n(x) := \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\{X_i \leq x\} \quad (x \in \mathbb{R}) \tag{3.4}
\]

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 \texttt{ECDF}, where

- A given sample \( \{X_i\}_{i=1}^n \) are the instance data, stored as \texttt{self.observations}
- The class implements a \texttt{__call__} method that returns \( F_n(x) \) for any \( x \)

Your code should work as follows (modulo randomness)

```python
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

0.29

F.observations = [uniform(0, 1) for i in range(1000)]
F(0.5)

0.479
```

Aim for clarity, not efficiency

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 \texttt{Polynomial} for representing and manipulating polynomial functions such as

\[
p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_N x^N = \sum_{n=0}^{N} a_n x^n \quad (x \in \mathbb{R}) \tag{3.5}
\]
The instance data for the class `Polynomial` will be the coefficients (in the case of (3.5), the numbers $a_0, \ldots, a_N$)

Provide methods that

1. Evaluate the polynomial (3.5), 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

### 3.2.6 Solutions

#### Exercise 1

```python
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)

# == 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))
```

```
0.5
0.486
```

#### Exercise 2

```python
class Polynomial:
    def __init__(self, coefficients):
        """
        Creates an instance of the Polynomial class representing
        """
```
\[ p(x) = a_0 x^0 + \ldots + a_N x^N, \]

where \( a_i = \text{coefficients}[i] \).

```python
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
```

### 3.3 OOP III: The Samuelson Accelerator

#### Contents

- OOP III: The Samuelson Accelerator
  - Overview
  - Details
  - Implementation
  - Stochastic shocks
  - Government spending
  - Wrapping everything into a class
  - Using the LinearStateSpace class
  - Pure multiplier model
  - Summary

Co-author: Natasha Watkins
3.3.1 Overview

This lecture creates nonstochastic and stochastic versions of Paul Samuelson’s celebrated multiplier accelerator model [Sam39].

In doing so, we extend the example of the Solow model class in our second OOP lecture.

Our objectives are to:

- provide a more detailed example of OOP and classes
- review a famous model
- review linear difference equations, both deterministic and stochastic

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 [Sar87])

Samuelson used the model to analyze how particular values of the marginal propensity to consume and the accelerator coefficient might give rise to transient business cycles in national output.

Possible dynamic properties include:

- smooth convergence to a constant level of output
- damped business cycles that eventually converge to a constant level of output
- persistent business cycles that neither dampen nor explode

Later we present an extension that adds a random shock to the right side of the national income identity representing random fluctuations in aggregate demand.

This modification makes national output become governed by a second-order stochastic linear difference equation that, with appropriate parameter values, gives rise to recurrent irregular business cycles.

(To read about stochastic linear difference equations see chapter XI of [Sar87])
3.3.2 Details

Let's assume that

- \( \{G_t\} \) is a sequence of levels of government expenditures. We'll start by setting \( G_t = G \) for all \( t \)
- \( \{C_t\} \) is a sequence of levels of aggregate consumption expenditures, a key endogenous variable in the model
- \( \{I_t\} \) is a sequence of rates of investment, another key endogenous variable
- \( \{Y_t\} \) is a sequence of levels of national income, yet another endogenous variable
- \( a \) is the marginal propensity to consume in the Keynesian consumption function \( C_t = aY_{t-1} + \gamma \)
- \( b \) is the accelerator coefficient in the investment accelerator \( I_t = b(Y_{t-1} - Y_{t-2}) \)
- \( \{\epsilon_t\} \) is an IID sequence of standard normal random variables
- \( \sigma \geq 0 \) is a volatility parameter; setting \( \sigma = 0 \) recovers the nonstochastic case that we'll start with

The model combines the consumption function

\[
C_t = aY_{t-1} + \gamma \tag{3.6}
\]

with the investment accelerator

\[
I_t = b(Y_{t-1} - Y_{t-2}) \tag{3.7}
\]

and the national income identity

\[
Y_t = C_t + I_t + G_t \tag{3.8}
\]

- The parameter \( a \) is people's marginal propensity to consume out of income - equation (3.6) 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 (3.7) asserts that people invest in physical capital when income is increasing and disinvest when it is decreasing

Equations (3.6), (3.7), and (3.8) 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

\[
Y_t = \rho_1Y_{t-1} + \rho_2Y_{t-2} + (\gamma + G_t) \tag{3.9}
\]

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 \( t = 0, \ldots, T \), we require initial values

\[
Y_{-1} = \bar{Y}_{-1}, \quad Y_{-2} = \bar{Y}_{-2}
\]

Well 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.

**Stochastic version of the model**

We create a **random** or **stochastic** version of the model by adding a random process of **shocks** or **disturbances** \( \{\sigma_t\} \) to the right side of equation (3.9), leading to the **second-order scalar linear stochastic difference equation**:

\[
Y_t = G_t + a(1 - b)Y_{t-1} - abY_{t-2} + \sigma_t
\]

(3.10)

**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 (3.10) as

\[
Y_t = \rho_1 Y_{t-1} + \rho_2 Y_{t-2}
\]

or

\[
Y_{t+2} - \rho_1 Y_{t+1} - \rho_2 Y_t = 0
\]

(3.11)

To discover the properties of the solution of (3.11), it is useful first to form the **characteristic polynomial** for (3.11):

\[
z^2 - \rho_1 z - \rho_2
\]

(3.12)

where \( z \) is possibly a complex number.
We want to find the two zeros (a.k.a. roots) – namely \( \lambda_1, \lambda_2 \) – 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 (3.12), the characteristic polynomial (3.12) equals zero:

\[
z^2 - \rho_1 z - \rho_2 = (z - \lambda_1)(z - \lambda_2) = 0
\]  

(3.13)

Equation (3.13) 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

\[
\lambda_1 = re^{i\theta}, \quad \lambda_2 = re^{-i\theta}
\]

where \( r \) is the amplitude of the complex number and \( \theta \) is its angle.

These can also be represented as

\[
\lambda_1 = r(\cos(\theta) + i\sin(\theta)) \\
\lambda_2 = r(\cos(\theta) - i\sin(\theta))
\]

(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 (3.11)

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 \( \rho_1, \rho_2 \)

When the roots are complex, some algebra that exploits the fact that the roots appear as a complex conjugate pair implies that

\[
Y_t = \tilde{c}_1 r^t \cos(\theta t + \tilde{c}_1)
\]

where \( \tilde{c}_1, \tilde{c}_2 \) is a pair of constants chosen to satisfy the given initial conditions for \( Y_{-1}, Y_{-2} \)

This formula shows that when the roots are complex, \( Y_t \) displays oscillations with period \( \tilde{p} = \frac{2\pi}{\theta} \) and damping factor \( r \)

Remark: Following [Sam39], we want to choose the parameters \( a, b \) of the model so that the absolute values (of the possibly complex) roots \( \lambda_1, \lambda_2 \) of the characteristic polynomial are both strictly less than one:

\[
|\lambda_j| < 1 \quad \text{for } j = 1, 2
\]

Remark: When both eigenvalues \( \lambda_1, \lambda_2 \) 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
**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 \( \lambda_1, \lambda_2 \) 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 periods \( Y_{t+1} \) 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 nonstochastic (\( \sigma = 0 \)) 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 (3.12) 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 \( G \):

\[
A = \begin{bmatrix}
1 & 0 & 0 \\
\gamma + G & \rho_1 & \rho_2 \\
0 & 1 & 0
\end{bmatrix}
\]

### 3.3.3 Implementation

Well start by drawing an informative graph from page 189 of [Sar87].
import numpy as np
import matplotlib.pyplot as plt

def param_plot():
    """this function creates the graph on page 189 of Sargent Macroeconomic Theory, second edition, 1987""

    fig, ax = plt.subplots(figsize=(12, 8))
    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'$\rho_1$', fontsize=16)  
    ax.set_ylabel(r'$\rho_2$', fontsize=16)  
    ax.set_label_position('top')
    ax.set_ylim(r'$\rho_1$', rotation=0, fontsize=16)

    # Draw (t1, t2) points
    rho_1 = np.linspace(-2, 2, 100)
    rho_1 = 1 + 2 / 4
    ax.plot(rho_1, -np.ones_like(rho_1) + 1, c='black')
    ax.plot(rho_1, -np.ones_like(rho_1) - 1, 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_arrowsl, fontsize='12')
    ax.annotate(r'$\rho_1 + \rho_2 = 1$', xy=(0.5, 0.3), xytext=(0.1, 0.6),
                arrowprops=plot_arrowsr, fontsize='12')
    ax.annotate(r'$\rho_2 < 1 + \sqrt{\rho_1}$', xy=(-0.5, 0.3), xytext=(-1.3, 0.6),
                arrowprops=plot_arrowsl, fontsize='12')
    ax.annotate(r'$\rho_2 = 1 + \sqrt{\rho_1}$', xy=(-0.5, 0.3), xytext=(-1, 0.8),
                arrowprops=plot_arrowsr, 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')
The graph portrays regions in which the \((\lambda_1, \lambda_2)\) root pairs implied by the \((\rho_1 = (a+b), \rho_2 = -b)\) difference equation parameter pairs in the Samuelson model are such that:

- \((\lambda_1, \lambda_2)\) are complex with modulus less than 1 - in this case, the \(\{Y_t\}\) sequence displays damped oscillations
• \((\lambda_1, \lambda_2)\) are both real, but one is strictly greater than 1 - this leads to explosive growth
• \((\lambda_1, \lambda_2)\) are both real, but one is strictly less than \(-1\) - this leads to explosive oscillations
• \((\lambda_1, \lambda_2)\) 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 will present the graph with a red mark showing the particular point implied by the setting of \((a, b)\)

Function to describe implications of characteristic polynomial

```python
def categorize_solution(\rho_1, \rho_2):
    """this function takes values of \(\rho_1\) and \(\rho_2\) and uses them to classify the type of solution"""

discriminant = \rho_1 ** 2 + 4 * \rho_2
if \rho_2 > 1 + \rho_1 or \rho_2 < -1:
    print('Explosive oscillations')
elif \rho_1 + \rho_2 > 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 zero; therefore get smooth convergence to a steady state')

### Test the categorize_solution function

categorize_solution(1.3, -.4)

Roots are real and absolute values are less than zero; therefore get smooth convergence to a steady state

Function for plotting \(Y_t\) paths

A useful function for our work below

```python
def plot_y(function=None):
    """function plots path of \(Y_t\)"""
    plt.subplots(figsize=(12, 8))
    plt.plot(function)
    plt.xlabel('Time \$t\$')
    plt.ylabel('\$Y_t\$', rotation=0)
    plt.grid()
    plt.show()
Manual or by hand root calculations

The following function calculates roots of the characteristic polynomial using high school algebra
(Well calculate the roots in other ways later)

The function also plots a $Y_t$ starting from initial conditions that we set

```python
from cmath import sqrt

### 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 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]
```

3.3. OOP III: The Samuelson Accelerator
for t in range(2, n):
    y_t.append(transition(y_t, t))
return y_t

plot_y(y_nonstochastic())

\(\rho_1 \text{ is } 1.42\)
\(\rho_2 \text{ is } -0.5\)

Two real roots:
\([-0.645687576256715, -0.77403123743284]\)

Absolute values of roots are less than one

**Reverse engineering parameters to generate damped cycles**

The next cell writes code that takes as inputs the modulus \(r\) and phase \(\phi\) 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 \((\rho_1, \rho_2)\), pairs that would generate those roots
### code to reverse engineer a cycle
### \( y_t = r^t (c_1 \cos(t) + c_2 \sin(t)) \)
###
```python
import cmath
import math

def f(r, ):
    """Takes modulus \( r \) and angle of complex number \( r \exp(j) \)
    and creates \( \rho_1 \) and \( \rho_2 \) 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, )  # Generate two complex roots
    g2 = cmath.rect(r, -)  # Implied \( \rho_1, \rho_2 \)
    \( \rho_1 = g1 + g2 \)
    \( \rho_2 = -g1 \times g2 \)
    b = -\rho2  # Reverse engineer \( a \) and \( b \) that validate these
    a = \rho1 - b
    return \rho1, \rho2, 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
    = 2 * math.pi/period

## Apply the function
\rho1, \rho2, a, b = f(r, )

print(f"a, b = {a}, {b}\")
print(f"\rho1, \rho2 = {\rho1}, {\rho2}\")
```

\[ a, b = (0.6346322893124001+0j) (0.9024999999999999-0j) \]
\[ \rho1, \rho2 = (1.5371322893124001+0j) (-0.9024999999999999+0j) \]

## Print the real components of \( \rho_1 \) and \( \rho_2 \)
\[ \rho1 = \rho1\text{.real} \]
\[ \rho2 = \rho2\text{.real} \]

\[ (1.5371322893124, -0.9024999999999999) \]

3.3. OOP III: The Samuelson Accelerator
Root finding using numpy

Here well use numpy to compute the roots of the characteristic polynomial

```python
r1, r2 = np.roots([1, -ρ1, -ρ2])
p1 = cmath.polar(r1)
p2 = cmath.polar(r2)

print(f"r, = {r1}, {r2}")
print(f"p1, p2 = {p1}, (p2)")
# print(f"g1, g2 = (g1), (g2)")
print(f"a, b = (a), (b)")
print(f"p1, ρ2 = (p1), (ρ2)")
```

r, = 0.95 0.6283185307179586
p1, p2 = (0.95, 0.6283185307179586) (0.95, -0.6283185307179586)
a, b = (0.6346322893124001+0j) (0.9024999999999999-0j)
ρ1, ρ2 = 1.5371322893124 -0.9024999999999999

```python
### This method uses numpy to calculate roots ===#
def y_nonstochastic(y_0=100, y_1=80, α=.9, β=.8, γ=10, n=80):
    """
    Rather than computing the roots of the characteristic polynomial by
    hand as we did earlier, this function
    enlists numpy to do the work for us ""
    
    # Useful constants
    ρ1 = α + β
    ρ2 = -β

    categorize_solution(ρ1, ρ2)

    # Find roots of polynomial
    roots = np.roots([1, -ρ1, -ρ2])
    print(f'Roots are {roots}')

    # Check if real or complex
    if all(isinstance(root, complex) for root in roots):
        print('Roots are complex')
    else:
        print('Roots are real')

    # Check if roots are less than one
    if all(abs(root) < 1 for root in roots):
        print('Roots are less than one')
    else:
        print('Roots are not less than one')
```

# Define transition equation
def transition(x, t):
    return ρ1 * x[t - 1] + ρ2 * x[t - 2] + γ

# Set initial conditions
y_t = [y_0, y_1]

# Generate y_t series
for t in range(2, n):
    y_t.append(transition(y_t, t))

return y_t

plot_y(y_nonstochastic())

Roots are complex with modulus less than one; therefore damped oscillations
Roots are [ 0.85+0.27838822j  0.85-0.27838822j]
Roots are complex
Roots are less than one

Reverse engineered complex roots: example

The next cell studies the implications of reverse engineered complex roots

Well generate an undamped cycle of period 10
\[ r = 1 \quad \# \text{generates undamped, nonexplosive cycles} \]
\[ \text{period} = 10 \quad \# \text{length of cycle in units of time} \]
\[ = 2 * \text{math.pi}/\text{period} \]

```python
## Apply the reverse engineering function f

\( \rho_1, \rho_2, a, b = f(r, ) \)

\[ a = a.\text{real} \quad \# \text{drop the imaginary part so that it is a valid input into y_{nonstochastic}} \]
\[ b = b.\text{real} \]

```

print(f"a, b = {a}, {b}"")
```

\[ y_{\text{temp}} = y_{\text{nonstochastic}}(a=a, \beta=b, y_0=20, y_1=30) \]

```
print(f"a, b = 0.6180339887498949 1.0"
Roots are complex with modulus less than one; therefore damped oscillations
Roots are \[ 0.80901699+0.58778525j \quad 0.80901699-0.58778525j \]
Roots are complex
Roots are less than one
```

\[ y_{\text{t}} \]
\[ y_{\text{c}} \]
\[ y_{\text{i}} \]
\[ y_{\text{j}} \]
\[ y_{\text{k}} \]
\[ y_{\text{l}} \]
\[ y_{\text{m}} \]
\[ y_{\text{n}} \]
\[ y_{\text{o}} \]
\[ y_{\text{p}} \]
\[ y_{\text{q}} \]
\[ y_{\text{r}} \]
\[ y_{\text{s}} \]
\[ y_{\text{t}} \]
\[ y_{\text{u}} \]
\[ y_{\text{v}} \]
\[ y_{\text{w}} \]
\[ y_{\text{x}} \]
\[ y_{\text{y}} \]
\[ y_{\text{z}} \]
\[ 10 \]
\[ 15 \]
\[ 20 \]
\[ 25 \]
\[ 30 \]
\[ 35 \]
\[ 40 \]
\[ 0 \]
\[ 10 \]
\[ 20 \]
\[ 30 \]
\[ 40 \]
\[ 50 \]
\[ 60 \]
\[ 70 \]
\[ 80 \]
\[ \text{Time } t \]
Digression: using sympy to find roots

We can also use sympy to compute analytic formulas for the roots

```python
import sympy
from sympy import Symbol, init_printing

init_printing()

r1 = Symbol("ρ_1")
r2 = Symbol("ρ_2")
z = Symbol("z")

sympy.solve(z**2 - r1*z - r2, z)
```

\[
\left[ \frac{\rho_1}{2} - \frac{1}{2} \sqrt{\rho_1^2 + 4\rho_2}, \quad \frac{\rho_1}{2} + \frac{1}{2} \sqrt{\rho_1^2 + 4\rho_2} \right]
\]

```python
a = Symbol("α")
b = Symbol("β")
r1 = a + b
r2 = -b

sympy.solve(z**2 - r1*z - r2, z)
```

\[
\left[ \frac{\alpha + \beta}{2} - \frac{1}{2} \sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}, \quad \frac{\alpha + \beta}{2} + \frac{1}{2} \sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta} \right]
\]

### 3.3.4 Stochastic shocks

Now well construct some code to simulate the stochastic version of the model that emerges when we add a random shock process to aggregate demand

```python
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):
```

3.3. OOP III: The Samuelson Accelerator
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
= np.random.normal(0, 1, n)

# Define transition equation
def transition(x, t):
    return \n        * x[t - 1] + \n          * x[t - 2] + \n        + [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 zero; 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

\[ r = 0.97 \]

period = 10  # length of cycle in units of time
  = 2 * math.pi / period

### apply the reverse engineering function \( f \)

\[ p_1, p_2, a, b = f(r, \) \]

\[ a = a.\text{real} \quad \text{# drop the imaginary part so that it is a valid input into } y_\text{nonstochastic} \]
\[ b = b.\text{real} \]

print(f"a, b = (a), (b)"
plot_y(y_stochastic(y_0=40, y_1=42, a=a, b=b, \sigma=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

3.3. OOP III: The Samuelson Accelerator
3.3.5 Government spending

This function computes a response to either a permanent or one-off increase in government expenditures

```python
def y_stochastic_g(y_0=20,
y_1=20,
alpha=0.8,
beta=0.2,
gamma=10,
n=100,
sigma=2,
g=0,
g_t=0,
duration='permanent'):
    
    # Useful constants
    rho1 = alpha + beta
    rho2 = -beta

    # Categorize solution
categorize_solution(rho1, rho2)
```

"""This program computes a response to a permanent increase in government expenditures that occurs at time 20"""

# Useful constants
rho1 = alpha + beta
rho2 = -beta

# Categorize solution
categorize_solution(rho1, rho2)
# Find roots of polynomial
roots = np.roots([1, -p1, -p2])
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
= 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 p1 * x[t - 1] + p2 * x[t - 2] + γ + g

    # Stochastic
    else:
        = np.random.normal(0, 1, n)
        return p1 * x[t - 1] + p2 * x[t - 2] + γ + g + σ * [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))

3.3. OOP III: The Samuelson Accelerator
# One-off government spending shock

def transition(y_t, t, g):
    return y_t + transition(y_t, t, g)

if t == g_t:
    y_t.append(transition(y_t, t, g))
else:
    y_t.append(transition(y_t, t, 0))

A permanent government spending shock can be simulated as follows:

plot_y(y_stochastic_g(g=10, g_t=20, duration='permanent'))

Roots are real and absolute values are less than zero; therefore get smooth convergence to a steady state:

[ 0.7236068 0.2763932]

Roots are real
Roots are less than one

We can also see the response to a one time jump in government expenditures:

plot_y(y_stochastic_g(g=500, g_t=50, duration='one-off'))
Roots are real and absolute values are less than zero; therefore get smooth convergence to a steady state

\[
\begin{bmatrix}
0.7236068 \\
0.2763932
\end{bmatrix}
\]

Roots are real
Roots are less than one

3.3.6 Wrapping everything into a class

Up to now we have written functions to do the work

Now well roll up our sleeves and write a Python class called *Samuelson* for the Samuelson model

```python
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}"
```

3.3. OOP III: The Samuelson Accelerator
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. Must be greater than or equal to 0. Set equal to 0 for 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):
    p1, p2 = self.p1, self.p2
    discriminant = p1 ** 2 + 4 * p2
    if p2 >= 1 + p1 or p2 <= -1:
        return 'Explosive oscillations'
    elif p1 + p2 >= 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.sigma == 0:
        return self.p1 * x[t - 1] + self.p2 * x[t - 2] + self.eta + g
    # Stochastic
    else:
        epsilon = np.random.normal(0, 1, self.n)
        return self.p1 * x[t - 1] + self.p2 * x[t - 2] + self.eta + g +
        self.sigma * epsilon

def generate_series(self):
    # Create list and set initial conditions
    y_t = [self.y0, self.y1]
    # 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))

3.3. OOP III: The Samuelson Accelerator
# One-off government spending shock

`elif self.duration == 'one-off':`

```python
    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`

```python
def summary(self):
    print('Summary')
    print('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=(12, 8))
    ax.plot(self.generate_series())
    ax.set(xlabel='Iteration', xlim=(0, self.n))
    ax.set_ylabel('$Y_t$', rotation=0)
    ax.grid()
    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)
```python
fig = param_plot()
ax = fig.gca()

# Add \lambda values to legend
for i, root in enumerate(self.roots):
    if isinstance(root, complex):
        operator = ['+', '']  # Need to fill operator for positive as string is split apart
        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 \rho pair to plot
ax.scatter(self.rh1, self.rh2, 100, 'red', '+', label=r'$(\ \rho_1, \ \rho_2)$', zorder=5)
plt.legend(fontsize=12, loc=3)

return fig
```

**Illustration of Samuelson class**

Now well put our Samuelson class to work on an example

```python
sam = Samuelson(a=0.8, b=0.5, \sigma=2, g=10, g_t=20, duration='permanent')
sam.summary()
```

<table>
<thead>
<tr>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root type: Complex conjugate</td>
</tr>
<tr>
<td>Solution type: Damped oscillations</td>
</tr>
<tr>
<td>Roots: [0.65+0.27838822j 0.65-0.27838822j]</td>
</tr>
<tr>
<td>Absolute value of roots is less than one</td>
</tr>
<tr>
<td>Stochastic series with \sigma = 2</td>
</tr>
<tr>
<td>Government spending equal to 10</td>
</tr>
<tr>
<td>Permanent government spending shock at t = 20</td>
</tr>
</tbody>
</table>

```python
sam.plot()
plt.show()
```
Using the graph

Well 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 $\pm$ sign shows the location of the roots.

```python
sam.param_plot()
plt.show()
```
3.3.7 Using the LinearStateSpace class

It turns out that we can use the QuantEcon.py LinearStateSpace class to do much of the work that we have done from scratch above.

Here is how we map the Samuelson model into an instance of a LinearStateSpace class:

```python
from quantecon import LinearStateSpace

""" This script maps the Samuelson model in the LinearStateSpace class"""
alpha = 0.8
beta = 0.9
rho1 = alpha + beta
rho2 = -beta
gamma = 10
sigma = 1
q = 10
n = 100

A = [[1, 0, 0],
     [gamma + g, rho1, rho2],
     [0, 1, 0]]

G = [[gamma + g, rho1, rho2],
     [gamma, alpha, 0]]  # this is Y_(t+1)
```

where $\lambda_1 = 0.65 + 0.28i$ and $\lambda_2 = 0.65 - 0.28i$. 

---

**3.3. OOP III: The Samuelson Accelerator**
\[0, \beta, -\beta]\]  # this is \(I_{t+1}\)

\[
\mu_0 = [1, 100, 100]
\]

\(C = np.zeros((3,1))\)

\(C[1] = \sigma \]  # stochastic

\(\text{sam}_t = \text{LinearStateSpace}(A, C, G, \mu_0=\mu_0)\)

\(x, y = \text{sam}_t.\text{simulate}(ts\text{-}length=n)\)

\(\text{fig, axes = plt.subplots(3, 1, sharex=True, figsize=(15, 8))}\)

\(\text{titles} = ['Output ($Y_t$)', 'Consumption ($C_t$)', 'Investment ($I_t$)']\)

\(\text{colors} = ['darkblue', 'red', 'purple']\)

\(\text{for ax, series, title, color in zip(axes, y, titles, colors):}\)

\(\text{ax.plot(series, color=color)}\)

\(\text{ax.set(title=title, xlim=(0, n))}\)

\(\text{ax.grid()}\)

\(\text{axes[-1].set_xlabel('Iteration')}\)

\(\text{plt.show()}\)

**Other methods in the LinearStateSpace class**

Lets plot **impulse response functions** for the instance of the Samuelson model using a method in the LinearStateSpace class
imres = sam_t.impulse_response()
imres = np.asarray(imres)
y1 = imres[:, :, 0]
y2 = imres[:, :, 1]
y1.shape

(2, 6, 1)

Now let's compute the zeros of the characteristic polynomial by simply calculating the eigenvalues of $A$

```python
A = np.asarray(A)
w, v = np.linalg.eig(A)
print(w)
```

```
[ 0.85+0.42130749j  0.85-0.42130749j  1.00+0.j]
```

**Inheriting methods from `LinearStateSpace`**

We could also create a subclass of `LinearStateSpace` (inheriting all its methods and attributes) to add more functions to use

```python
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,
        alpha=0.8,
        beta=0.9,
        gamma=10,
        sigma=1,
        g=10):
        self.alpha, self.beta = alpha, beta
        self.y_0, self.y_1, self.g = y_0, y_1, g
        self.gamma, self.sigma = gamma, sigma

        # Define initial conditions
        self.m_0 = [1, y_0, y_1]

        self.r_1 = alpha + beta
        self.r_2 = -beta

        # Define transition matrix
        self.A = [[1, 0, 0],
                  [gamma + g, self.r_1, self.r_2],
                  [0, 1, 0]]

```

3.3. OOP III: The Samuelson Accelerator
# Define output matrix
self.G = [[γ + g, self.r1, self.r2],
          [γ, α, 0],
          [0, β, -β]]  # this is Y_{t+1}
# this is C_{t+1}
# this is I_{t+1}

self.C = np.zeros((3, 1))
self.C[1] = σ  # stochastic

# Initialize LSS with parameters from Samuleson model
LinearStateSpace.__init__(self, self.A, self.C, self.G, mu_0=self._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=(15, 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, xtickl=(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=(15, 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)

Illustrations

Lets show how we can use the *SamuelsonLSS*

samlss = SamuelsonLSS()
samlss.plot_simulation(100, stationary=False)
plt.show()
```
samlss.plot_simulation(100, stationary=True)
plt.show()
```

```
samlss.plot_irf(100)
plt.show()
```
3.3.8 Pure multiplier model

Let’s shut down the accelerator by setting $b = 0$ to get a pure multiplier model

- the absence of cycles gives an idea about why Samuelson included the accelerator

```python
samlss.multipliers()
array([[ 7.414389,  6.835896,  0.578493]])
```

```python
pure_multiplier = SamuelsonLSS($\alpha=0.95$, $\beta=0$)
pure_multiplier.plot_simulation()
Stationary distribution does not exist
```
pure_multiplier = SamuelsonLSS(α=0.8, β=0)

pure_multiplier.plot_simulation()
3.3.9 Summary

In this lecture, we wrote functions and classes to represent non-stochastic and stochastic versions of the Samuelson (1939) multiplier-accelerator model, described in [Sam39].

We saw that different parameter values led to different output paths, which could either be stationary, explosive, or oscillating.

We also were able to represent the model using the QuantEcon.py LinearStateSpace class.

3.4 More Language Features

Contents

- More Language Features
  - Overview
  - Iterables and Iterators
  - Names and Name Resolution
  - Handling Errors
  - Decorators and Descriptors
3.4.1 Overview

With this last lecture, our advice is to skip it on first pass, unless you have a burning desire to read it. It is 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.

3.4.2 Iterables and Iterators

We’ve already said something about iterating in Python. Now let’s look more closely at how it all works, focusing on Python’s implementation of the for loop.

**Iterators**

Iterators are a uniform interface to stepping through elements in a collection. Here we’ll talk about using iterators later, and 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*.

```python
f = open('us_cities.txt')
f.__next__()

'new york: 8244910\n'
```

```python
f.__next__()

'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

```python
next(f)
```

'chicago: 2707120 

The objects returned by `enumerate()` are also iterators

```python
e = enumerate(['foo', 'bar'])
next(e)
```

(0, 'foo')

```python
next(e)
```

(1, 'bar')

as are the reader objects from the `csv` module

```python
from csv import reader
f = open('test_table.csv', 'r')
nikkei_data = reader(f)
next(nikkei_data)
```

[['Date', 'Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close']

```python
next(nikkei_data)
```

[['2009-05-21', '9280.35', '9286.35', '9189.92', '9264.15', '133200', '9264.15 →']

**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

```python
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

### 3.4. More Language Features
The interpreter just keeps
1. calling \texttt{f.__next__()} and binding \texttt{line} to the result
2. executing the body of the loop

This continues until a \texttt{StopIteration} error occurs

\textbf{Iterables}

You already know that we can put a Python list to the right of \texttt{in} in a for loop

\begin{verbatim}
for i in ['spam', 'eggs']:
    print(i)
\end{verbatim}

\texttt{spam}
\texttt{eggs}

So does that mean that a list is an iterator?

The answer is no:

\begin{verbatim}
x = ['foo', 'bar']
type(x)
\end{verbatim}

\texttt{list}

\begin{verbatim}
next(x)
\end{verbatim}

```
TypeError                                 Traceback (most recent call last)
<ipython-input-17-5e4e57af3a97> 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 \textit{iterable} (as opposed to an iterator)

Formally, an object is iterable if it can be converted to an iterator using the built-in function \texttt{iter()}

Lists are one such object

\begin{verbatim}
x = ['foo', 'bar']
type(x)
\end{verbatim}
Many other objects are iterable, such as dictionaries and tuples

Of course, not all objects are 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

Iterators and built-ins

Some built-in functions that act on sequences also work with iterables

- max(), min(), sum(), all(), any()
For example

\[
x = [10, -10]
\]
\[\text{max}(x)\]

10

\[
y = \text{iter}(x)
\]
\[\text{type}(y)\]

listiterator

\[\text{max}(y)\]

10

One thing to remember about iterators is that they are depleted by use

\[
x = [10, -10]
\]
\[y = \text{iter}(x)
\]
\[\text{max}(y)\]

\[
\text{max}(y)
\]

```
ValueError
Traceback (most recent call last)
<ipython-input-72-1d3b6314f310> in <module>()
      ----> 1 max(y)
ValueError: max() arg is an empty sequence
```

3.4.3 Names and Name Resolution

Variable Names in Python

Consider the Python statement

\[
x = 42
\]

We now know that when this statement is executed, Python creates an object of type \text{int} in your computers 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.

```python
def f(string):
    # Create a function called f
    print(string)  # that prints any string it’s passed

g = f
id(g) == id(f)
```

```
True
```

```
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.

```python
x = 'foo'
id(x)
```

```
164994764
```

```python
x = 'bar'  # No names bound to object 164994764
```

What happens here is that the first object, with identity 164994764 is garbage collected.

In other words, the memory slot that stores that object is deallocated, and returned to the operating system.

**Namespaces**

Recall from the preceding discussion that the statement

```python
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` like this

```python
# Filename: math2.py
pi = 'foobar'
```

Now we start the Python interpreter and import it

```python
import math2
```

Next let's import the `math` module from the standard library

```python
import math
```

Both of these modules have an attribute called `pi`

```python
math.pi
```

```python
3.1415926535897931
```

```python
math2.pi
```

```python
'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__`

```python
import math
math.__dict__
```

```python
({'pow': <built-in function pow>, ..., 'pi': 3.1415926535897931,...} # Edited output
```

```python
import math2
math2.__dict__
```

```python
(..., '__file__': 'math2.py', 'pi': 'foobar',...) # Edited output
```

As you know, we access elements of the namespace using the dotted attribute notation

```python
math.pi
```
3.1415926535897931

In fact this is entirely equivalent to `math.__dict__['pi']`

```python
math.__dict__['pi'] == math.pi
```

True

**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)`

```python
vars(math)
```

```python
{'pow': <built-in function pow>,...
```

If you just want to see the names, you can type

```python
dir(math)
```

```python
['__doc__', '__name__', 'acos', 'asin', 'atan',...
```

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

```python
print(math.__doc__)
```

This module is always available. It provides access to the mathematical functions defined by the C standard.

```python
math.__name__
```

'math'

**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

```python
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

```python
# Filename: mod.py
print(__name__)
```

Now let's look at two different ways of running it in IPython

```python
import mod  # Standard import
mod
%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`

```python
x = 2
y = 3
import numpy as np
%whos
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Data/Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>np</td>
<td>module</td>
<td>&lt;module 'numpy' from '/usr&lt;...&gt;ages/numpy/<strong>init</strong>.pyc'&gt;</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>y</td>
<td>int</td>
<td>3</td>
</tr>
</tbody>
</table>

### 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`

```python
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 its `__main__`, so the namespace of `__main__` again becomes the global namespace

**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

```python
def f(x):
    a = 2
    print(locals())
    return a * x
```

Now lets call the function

```python
f(1)
```

```python
{'a': 2, 'x': 1}
```

You can see the local namespace of `f` before it is destroyed

**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?

3.4. More Language Features 219
• These definitions are stored in a module called __builtin__
• They have there own namespace called __builtins__

```
dir()
```

```
[... , '__builtins__', '__doc__', ...] # Edited output
```

```
dir(__builtins__)
```

```
[... 'iter', 'len', 'license', 'list', 'locals', ...] # Edited output
```

We can access elements of the namespace as follows

```
__builtins__.max
```

```
<built-in function max>
```

But __builtins__ is special, because we can always access them directly as well

```
max
```

```
<built-in function max>
```

```
__builtins__.max == max
```

```
True
```

The next section explains how this works

**Name Resolution**

Namespaces are great because they help us organize variable names

(Type import this at the prompt and look at the last item thats 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

```python
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

```python
def g(x):
    a = 1
    x = x + a
    return x

a = 0
y = g(10)
print("a = ", a, "y = ", y)
```

What happens when we run this script?

```
% run test.py
```

```
a = 0 y = 11
```

```
```

```
NameError Traceback (most recent call last)
<ipython-input-2-401b30e3b8b5> in <module>() ----> 1 x
NameError: name 'x' is not defined
```

**3.4. More Language Features**
First,

1. The global namespace `{}` is created
2. The function object is created, and `g` is bound to it within the global namespace
3. 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

1. The local namespace for the function is created
2. Local names `x` and `a` are bound, so that the local namespace becomes `{ 'x': 10, 'a': 1 }
3. Statement `x = x + a` uses the local `a` and local `x` to compute `x + a`, and binds local name `x` to the result
4. This value is returned, and `y` is bound to it in the global namespace
5. Local `x` and `a` are discarded (and the local namespace is deallocated)

Note that the global `a` was not affected by the local `a`

**Mutable Versus Immutable Parameters**

This is a good time to say a little more about mutable vs immutable objects

Consider the code segment

```python
def f(x):
    x = x + 1
    return x

x = 1
print(f(x), x)
```

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, its a different story when we use a **mutable** data type such as a list

```python
def f(x):
    x[0] = x[0] + 1
    return x

x = [1]
print(f(x), x)
```
This prints \([2]\) as the value of \(f(\times)\) and \textit{same} for \(\times\)

Here’s what happens

- \(\times\) is registered as a function in the global namespace
- \(\times\) bound to \([1]\) in the global namespace
- The call \(f(\times)\)
  - Creates a local namespace
  - Adds \(\times\) to local namespace, bound to \([1]\)
  - The list \([1]\) is modified to \([2]\)
  - Returns the list \([2]\)
  - The local namespace is deallocated, and local \(\times\) is lost
- Global \(\times\) has been modified

### 3.4.4 Handling Errors

Sometimes it’s possible to anticipate errors as we’re writing code.

For example, the unbiased sample variance of sample \(y_1, \ldots, y_n\) is defined as

\[
s^2 := \frac{1}{n-1} \sum_{i=1}^{n} (y_i - \bar{y})^2 \quad \bar{y} = \text{sample mean}
\]

This can be calculated in NumPy using \texttt{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 reduces confidence in your code on the part of your users (if you are writing for others).

### Assertions

A relatively easy way to handle checks is with the \texttt{assert} keyword.

For example, pretend for a moment that the \texttt{np.var} function doesn’t exist and we need to write our own.
```python
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

```python
var([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

**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

Lets look at how this is done

**Exceptions**

Heres an example of a common error type

```python
def f:
```

```
File "<ipython-input-5-f5bdb6d29788>", line 1
  def f:
^  SyntaxError: invalid syntax
```

Since illegal syntax cannot be executed, a syntax error terminates execution of the program

Heres a different kind of error, unrelated to syntax
1 / 0

ZeroDivisionError

ZeroDivisionError: integer division or modulo by zero

Here's another

x1 = y1

NameError

NameError: name 'y1' is not defined

And another

'foo' + 6

TypeError

TypeError: cannot concatenate 'str' and 'int' objects

And another

X = []
x = X[0]

IndexError

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:

```python
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:

```python
f(2)
0.5
f(0)
Error: division by zero. Returned None
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.

```python
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:

```python
f(2)
0.5
f(0)
Error: division by zero. Returned None
```
If we feel lazy we can catch these errors together:

```python
def f(x):
    try:
        return 1.0 / x
    except (TypeError, ZeroDivisionError):
        print('Error: Unsupported operation. Returned None')
    return None
```

Here’s what happens:

```python
f(2)
```

0.5

```python
f(0)
```

Error: Unsupported operation. Returned None

```python
f('foo')
```

Error: Unsupported operation. Returned None

If we feel extra lazy we can catch all error types as follows:

```python
def f(x):
    try:
        return 1.0 / x
    except:
        print('Error. Returned None')
    return None
```

In general it’s better to be specific.

### 3.4.5 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.
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

```python
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.

```python
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:

```python
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)
```

This looks complicated so let's work through it slowly.

To unravel the logic, consider what happens when we say \( f = \text{check\_nonneg}(f) \).

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 nonnegative 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

```python
def f(x):
    return np.log(np.log(x))
```

They may well miss the line `f = \text{check\_nonneg}(f)`.

For this and other reasons, decorators were introduced to Python.
With decorators, we can replace the lines

```python
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

```python
@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

**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

```python
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 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:

```python
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
```

### 3.4. More Language Features
First let's check that we get the desired behavior

```python
car = Car()
car.miles = 1000
car.miles = 6000
car.kms
```

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 built-in 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 it's 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.

```python
class Car:
    def __init__(self, miles=1000):
        self._miles = miles
        self._kms = miles * 1.61

    @property
def miles(self):
        return self._miles
```
We won't go through all the details here.
For further information you can refer to the descriptor documentation.

### 3.4.6 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.

#### 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:

```python
singular = ('dog', 'cat', 'bird')
type(singular)
```

```text
tuple
```

```python
plural = [string + 's' for string in singular]
plural
```

```text
['dogs', 'cats', 'birds']
```

```python
type(plural)
```

```text
list
```

And here is the generator expression.
singular = ('dog', 'cat', 'bird')
plural = (string + 's' for string in singular)
type(plural)
gen = plural
next(gen)
'dogs'
next(gen)
'cats'
next(gen)
'birds'

Since `sum()` can be called on iterators, we can do this

```python
sum((x * x for x in range(10)))
```

285

The function `sum()` calls `next()` to get the items, adds successive terms

In fact, we can omit the outer brackets in this case

```python
sum(x * x for x in range(10))
```

285

**Generator Functions**

The most flexible way to create generator objects is to use generator functions

Lets look at some examples

**Example 1**

Here's a very simple example of a generator function

```python
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:

```python
def f():
    yield 'start'
    yield 'middle'
    # This line!
    yield 'end'
```

```python
type(f)
function

gen = f()
gen

<generator object f at 0x3b66a50>

next(gen)

'start'

next(gen)

'middle'

next(gen)

'end'

next(gen)
```

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.
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

```python
def g(x):
    while x < 100:
        yield x
        x = x * x
```

Let's see how it works

```python
g
<function __main__.g>
gen = g(2)
type(gen)
generator
next(gen)
2
next(gen)
4
next(gen)
16
next(gen)
```

---

StopIteration
Traceback (most recent call last)
<ipython-input-32-b2c61ce5e131> in <module>()
----> 1 gen.next()
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*

```python
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

```python
def g(x):
    while 1:
        yield x
        x = x * x
```

### 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

```python
import random

n = 10000000
draws = [random.uniform(0, 1) < 0.5 for i in range(n)]
sum(draws)
```

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

```python
n = 1000000000
draws = [random.uniform(0, 1) < 0.5 for i in range(n)]
```

```
---------------------------------------------------------------------------
MemoryError                       Traceback (most recent call last)
<ipython-input-9-20d1ec1dae24> in <module>()
----> 1 draws = [random.uniform(0, 1) < 0.5 for i in range(n)]
```

3.4. More Language Features
We can avoid these problems using iterators

Here is the generator function

```python
def f(n):
    i = 1
    while i <= n:
        yield random.uniform(0, 1) < 0.5
        i += 1
```

Now let’s do the sum

```python
n = 10000000
draws = f(n)
draws
```

```
<generator object at 0xb7d8b2cc>
```

```python
sum(draws)
```

4999141

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

### 3.4.7 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

$$x_{t+1} = 2x_t, \quad x_0 = 1$$

(3.14)

Obviously the answer is $2^t$

We can compute this easily enough with a loop

```python
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
```python
def x(t):
    if t == 0:
        return 1
    else:
        return 2 * x(t-1)
```

What happens here is that each successive call uses its 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.

Well meet less contrived applications of recursion later on.

### 3.4.8 Exercises

#### Exercise 1

The Fibonacci numbers are defined by

\[
x_{t+1} = x_t + x_{t-1}, \quad x_0 = 0, \quad x_1 = 1
\]  \hspace{1cm} (3.15)

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 \( t \)

#### Exercise 2

Complete the following code, and test it using this csv file, which we assume that youve put in your current working directory.

```python
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)
```

3.4. More Language Features
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

### 3.4.9 Solutions

**Exercise 1**

Here's the standard solution

```python
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

```python
print([x(i) for i in range(10)])
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

**Exercise 2**

A small sample from `test_table.csv` is included (and saved) in the code below for convenience

```bash
%%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
```
One solution is as follows

```python
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
```

<table>
<thead>
<tr>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-05-21</td>
</tr>
<tr>
<td>2009-05-20</td>
</tr>
<tr>
<td>2009-05-19</td>
</tr>
<tr>
<td>2009-05-18</td>
</tr>
<tr>
<td>2009-05-15</td>
</tr>
<tr>
<td>2009-05-14</td>
</tr>
<tr>
<td>2009-05-13</td>
</tr>
<tr>
<td>2009-05-12</td>
</tr>
<tr>
<td>2009-05-11</td>
</tr>
</tbody>
</table>

**Exercise 3**

Let's save the data first

```bash
% file numbers.txt
prices
3
8
7
21
```
Writing numbers.txt

```python
f = open('numbers.txt')
total = 0.0
for line in f:
    try:
        total += float(line)
    except ValueError:
        pass
f.close()
print(total)
```

39.0

### 3.5 Debugging

#### Contents

- Debugging
  - Overview
  - Debugging
  - Other Useful Magics

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

#### 3.5.1 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 focus on Jupyter and leave you to explore other settings
3.5.2 Debugging

The debug Magic

Lets consider a simple (and rather contrived) example

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

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
```

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)

Here's what happens when we run the code:

```
---------------------------------------------------------------------------
AttributeError               Traceback (most recent call last)
<ipython-input-1-ef5c75a58138> in <module>()
     8     plt.show()
     9
---> 10 plot_log()  # Call the function, generate plot

<ipython-input-1-ef5c75a58138> in plot_log()
     5       fig, ax = plt.subplots(2, 1)
     6       x = np.linspace(1, 2, 10)
----> 7       ax.plot(x, np.log(x))
     8     plt.show()
     9

AttributeError: 'numpy.ndarray' object has no attribute 'plot'
```

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:

```
---------------------------------------------------------------------------
NameError                   Traceback (most recent call last)
```

3.5. Debugging
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:

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

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: 'numpy.ndarray' object has no attribute 'plot'
```

But this time we type in the following cell block:

```bash
%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)

Its 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

Continue execution, only stop when a breakpoint is encountered.

Setting a Break Point

The preceding approach is handy but sometimes insufficient

Consider the following modified version of our function above

```python
import numpy as np
import matplotlib.pyplot as plt
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 weve accidentally written `np.logspace(1, 2, 10)` instead of `np.linspace(1, 2, 10)`

Now there wont be any exception, but the plot wont 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 the line `from IPython.core.debugger import Tracer; Tracer()()` inside the function code block

```python
import numpy as np
import matplotlib.pyplot as plt
from IPython.core.debugger import Pdb

def plot_log():
    Pdb().set_trace()
    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

```
> %run script.py
5 def plot_log():
  6     from IPython.core.debugger import Tracer; Tracer()()
----> 7     fig, ax = plt.subplots()
  8     x = np.logspace(1, 2, 10)
  9     ax.plot(x, np.log(x))
 10     plt.show()
```

```
ipdb> n
> %run script.py
5 from IPython.core.debugger import Tracer; Tracer()()
6 fig, ax = plt.subplots()
----> 7 x = np.logspace(1, 2, 10)
8     ax.plot(x, np.log(x))
9     plt.show()
```

```
ipdb> n
> %run script.py
6 fig, ax = plt.subplots()
7 x = np.logspace(1, 2, 10)
----> 8     ax.plot(x, np.log(x))
9     plt.show()
```

```
ipdb> 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`
3.5.3 Other Useful Magics

In this lecture we used the \texttt{\%debug} IPython magic

There are many other useful magics:

- \texttt{\%precision 4} sets printed precision for floats to 4 decimal places
- \texttt{\%whos} gives a list of variables and their values
- \texttt{\%quickref} gives a list of magics

The full list of magics is \texttt{here}
This part of the course provides a set of lectures focused on Data and Empirics using Python.

### 4.1 Pandas

This section covers the Pandas library for Python, which is a powerful tool for data manipulation and analysis. The contents include:

- Overview
- Series
- DataFrames
- On-Line Data Sources
- Exercises
- Solutions

#### 4.1.1 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 and SAS, 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\(^1\)
   - 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

```
import pandas as pd
import numpy as np
```

\(^1\) Wikipedia defines munging as cleaning data from one raw form into a structured, purged one.
4.1.2 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.

```python
s = pd.Series(np.random.randn(4), name='daily returns')
```

```
  0   0.4303
  1   0.6173
  2  -0.2654
  3  -0.8361
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.

```python
s * 100
```

```
  0   43.03
  1   61.73
  2  -26.54
  3  -83.61
Name: daily returns, dtype: float64
```

```python
np.abs(s)
```

```
  0   0.4303
  1   0.6173
  2   0.2654
  3   0.8361
Name: daily returns, dtype: float64
```

But Series provide more than NumPy arrays.

Not only do they have some additional (statistically oriented) methods.

```python
s.describe()
```

```
   count  mean     std       min  25%   50%  75%     max
  4.0000 -0.0134  0.6671 -0.8361 -0.4081 0.0824 0.4081
```

4.1. Pandas
But their indices are more flexible

```python
s.index = ['AMZN', 'AAPL', 'MSFT', 'GOOG']
s
AMZN  0.430271
AAPL  0.617328
MSFT -0.265421
GOOG -0.836113
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—this case, floats)

In fact, you can use much of the same syntax as Python dictionaries

```python
s['AMZN']
```

0.43027108469945924

```python
s['AMZN'] = 0
s
```

AMZN  0.000000
AAPL  0.617328
MSFT -0.265421
GOOG -0.836113
Name: daily returns, dtype: float64

```python
'AAPL' in s
```

True

### 4.1.3 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`, and can be downloaded [here](https://example.com).

Here’s the contents of `test_pwt.csv`
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:

```python
df = pd.read_csv('https://github.com/QuantEcon/QuantEcon.lectures.code/raw/master/pandas/data/test_pwt.csv')
type(df)
pandas.core.frame.DataFrame
df
```

We can select particular rows using standard Python array slicing notation.
To select columns, we can pass a list containing the names of the desired columns represented as strings:

```python
df[['country', 'tcgdp']]
```

To select both rows and columns using integers, the `iloc` attribute should be used with the format `.iloc[rows,columns]

```python
df.iloc[2:5, 0:4]
```

To select rows and columns using a mixture of integers and labels, the `loc` attribute can be used in a similar way:

```python
df.loc[df.index[[2:5]], ['country', 'tcgdp']]```

Let's imagine that were only interested in population 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.
df = df[['country','POP','tcgdp']]
df

<table>
<thead>
<tr>
<th></th>
<th>country</th>
<th>POP</th>
<th>tcgdp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Argentina</td>
<td>37335.653</td>
<td>295072.218690</td>
</tr>
<tr>
<td>1</td>
<td>Australia</td>
<td>19053.186</td>
<td>541804.652100</td>
</tr>
<tr>
<td>2</td>
<td>India</td>
<td>1006300.297</td>
<td>1728144.374800</td>
</tr>
<tr>
<td>3</td>
<td>Israel</td>
<td>6114.570</td>
<td>129253.894230</td>
</tr>
<tr>
<td>4</td>
<td>Malawi</td>
<td>11801.505</td>
<td>5026.221784</td>
</tr>
<tr>
<td>5</td>
<td>South Africa</td>
<td>45064.098</td>
<td>227242.369490</td>
</tr>
<tr>
<td>6</td>
<td>United States</td>
<td>282171.957</td>
<td>9898700.000000</td>
</tr>
<tr>
<td>7</td>
<td>Uruguay</td>
<td>3219.793</td>
<td>25255.961693</td>
</tr>
</tbody>
</table>

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

```python
df = df.set_index('country')
df
```

<table>
<thead>
<tr>
<th></th>
<th>POP</th>
<th>tcgdp</th>
</tr>
</thead>
<tbody>
<tr>
<td>country</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Argentina</td>
<td>37335.653</td>
<td>295072.218690</td>
</tr>
<tr>
<td>Australia</td>
<td>19053.186</td>
<td>541804.652100</td>
</tr>
<tr>
<td>India</td>
<td>1006300.297</td>
<td>1728144.374800</td>
</tr>
<tr>
<td>Israel</td>
<td>6114.570</td>
<td>129253.894230</td>
</tr>
<tr>
<td>Malawi</td>
<td>11801.505</td>
<td>5026.221784</td>
</tr>
<tr>
<td>South Africa</td>
<td>45064.098</td>
<td>227242.369490</td>
</tr>
<tr>
<td>United States</td>
<td>282171.957</td>
<td>9898700.000000</td>
</tr>
<tr>
<td>Uruguay</td>
<td>3219.793</td>
<td>25255.961693</td>
</tr>
</tbody>
</table>

Let's give the columns slightly better names

```python
df.columns = 'population', 'total GDP'
df
```

<table>
<thead>
<tr>
<th></th>
<th>population</th>
<th>total GDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>country</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Argentina</td>
<td>37335.653</td>
<td>295072.218690</td>
</tr>
<tr>
<td>Australia</td>
<td>19053.186</td>
<td>541804.652100</td>
</tr>
<tr>
<td>India</td>
<td>1006300.297</td>
<td>1728144.374800</td>
</tr>
<tr>
<td>Israel</td>
<td>6114.570</td>
<td>129253.894230</td>
</tr>
<tr>
<td>Malawi</td>
<td>11801.505</td>
<td>5026.221784</td>
</tr>
<tr>
<td>South Africa</td>
<td>45064.098</td>
<td>227242.369490</td>
</tr>
<tr>
<td>United States</td>
<td>282171.957</td>
<td>9898700.000000</td>
</tr>
<tr>
<td>Uruguay</td>
<td>3219.793</td>
<td>25255.961693</td>
</tr>
</tbody>
</table>

Population is in thousands, let’s revert to single units

```python
df['population'] = df['population'] * 1e3
df
```

4.1. Pandas
<table>
<thead>
<tr>
<th>country</th>
<th>population</th>
<th>total GDP</th>
<th>GDP percap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argentina</td>
<td>37335653</td>
<td>295072.218690</td>
<td>7903.229085</td>
</tr>
<tr>
<td>Australia</td>
<td>19053186</td>
<td>541804.652100</td>
<td>28436.433261</td>
</tr>
<tr>
<td>India</td>
<td>1006300297</td>
<td>1728144.374800</td>
<td>1717.324719</td>
</tr>
<tr>
<td>Israel</td>
<td>6114570</td>
<td>129253.894230</td>
<td>21138.672749</td>
</tr>
<tr>
<td>Malawi</td>
<td>11801505</td>
<td>5026.221784</td>
<td>425.896679</td>
</tr>
<tr>
<td>South Africa</td>
<td>45064098</td>
<td>227242.369490</td>
<td>5042.647686</td>
</tr>
<tr>
<td>United States</td>
<td>282171957</td>
<td>9898700.000000</td>
<td>35080.381854</td>
</tr>
<tr>
<td>Uruguay</td>
<td>3219793</td>
<td>25255.961693</td>
<td>7843.970620</td>
</tr>
</tbody>
</table>

Next we are going to add a column showing real GDP per capita, multiplying by 1,000,000 as we go because total GDP is in millions.

```python
df['GDP percap'] = df['total GDP'] * 1e6 / df['population']
df
```

<table>
<thead>
<tr>
<th>country</th>
<th>population</th>
<th>total GDP</th>
<th>GDP percap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argentina</td>
<td>37335653</td>
<td>295072.218690</td>
<td>7903.229085</td>
</tr>
<tr>
<td>Australia</td>
<td>19053186</td>
<td>541804.652100</td>
<td>28436.433261</td>
</tr>
<tr>
<td>India</td>
<td>1006300297</td>
<td>1728144.374800</td>
<td>1717.324719</td>
</tr>
<tr>
<td>Israel</td>
<td>6114570</td>
<td>129253.894230</td>
<td>21138.672749</td>
</tr>
<tr>
<td>Malawi</td>
<td>11801505</td>
<td>5026.221784</td>
<td>425.896679</td>
</tr>
<tr>
<td>South Africa</td>
<td>45064098</td>
<td>227242.369490</td>
<td>5042.647686</td>
</tr>
<tr>
<td>United States</td>
<td>282171957</td>
<td>9898700.000000</td>
<td>35080.381854</td>
</tr>
<tr>
<td>Uruguay</td>
<td>3219793</td>
<td>25255.961693</td>
<td>7843.970620</td>
</tr>
</tbody>
</table>

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.

```python
import matplotlib.pyplot as plt

df['GDP percap'].plot(kind='bar')
plt.show()
```

The following figure is produced.
At the moment the data frame is ordered alphabetically on the countries let's change it to GDP per capita

```python
df = df.sort_values(by='GDP per cap', ascending=False)
df
```

<table>
<thead>
<tr>
<th>country</th>
<th>population</th>
<th>total GDP</th>
<th>GDP per cap</th>
</tr>
</thead>
<tbody>
<tr>
<td>United States</td>
<td>282171957</td>
<td>9898700.000000</td>
<td>35080.381854</td>
</tr>
<tr>
<td>Australia</td>
<td>19053186</td>
<td>541804.652100</td>
<td>28436.433261</td>
</tr>
<tr>
<td>Israel</td>
<td>6114570</td>
<td>129253.894230</td>
<td>21138.672749</td>
</tr>
<tr>
<td>Argentina</td>
<td>37335653</td>
<td>295072.218690</td>
<td>7903.229085</td>
</tr>
<tr>
<td>Uruguay</td>
<td>3219793</td>
<td>25255.961693</td>
<td>7843.970620</td>
</tr>
<tr>
<td>South Africa</td>
<td>45064098</td>
<td>227242.369490</td>
<td>5042.647686</td>
</tr>
<tr>
<td>India</td>
<td>1006300297</td>
<td>1728144.374800</td>
<td>1717.324719</td>
</tr>
<tr>
<td>Malawi</td>
<td>11801505</td>
<td>5026.221784</td>
<td>425.896679</td>
</tr>
</tbody>
</table>

Plotting as before now yields

```python
df['GDP per cap'].plot(kind='bar')
plt.show()
```
4.1.4 On-Line Data Sources

Python makes it straightforward to query on line 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
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:

```python
import requests
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 using the `source` object returned by the call `requests.get('http://research.stlouisfed.org/fred2/series/UNRATE/downloaddata/UNRATE.csv')`

```python
url = 'http://research.stlouisfed.org/fred2/series/UNRATE/downloaddata/UNRATE.csv'
source = requests.get(url).content.decode().split('
')
source[0]
'DATE,VALUE\r\n'
source[1]
'1948-01-01,3.4\r\n'
source[2]
'1948-02-01,3.8\r\n'
```

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.

```python
data = pd.read_csv(url, index_col=0, parse_dates=True)
```

### 4.1. Pandas
The data has been read into a pandas DataFrame called `data` that we can now manipulate in the usual way

```python
type(data)
```

```
pandas.core.frame.DataFrame
```

```python
data.head()  # A useful method to get a quick look at a data frame
```

<table>
<thead>
<tr>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
</tr>
<tr>
<td>1948-01-01</td>
</tr>
<tr>
<td>1948-02-01</td>
</tr>
<tr>
<td>1948-03-01</td>
</tr>
<tr>
<td>1948-04-01</td>
</tr>
<tr>
<td>1948-05-01</td>
</tr>
</tbody>
</table>

```python
data.describe()  # Your output might differ slightly
```

```
VALUE
count  830.0
mean   5.8
std    1.6
min    2.5
25%    4.7
50%    5.6
75%    6.9
max    10.8
```

We can also plot the unemployment rate from 2006 to 2012 as follows

```python
data['2006':'2012'].plot()
plt.show()
```

The resulting figure looks as follows
Accessing World Bank Data

Let's look at one more example of downloading and manipulating 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.

If you click on DOWNLOAD DATA you will be given the option to download the data as an Excel file.

The next program does this for you, reads an Excel file into a pandas DataFrame, and plots time series for the US and Australia.

```python
import matplotlib.pyplot as plt
import requests
import pandas as pd

# == Get data and read into file gd.xls == #
r = requests.get(wb_data_query)
with open('gd.xls', 'wb') as output:
    output.write(r.content)
```

4.1. Pandas
# == Parse data into a DataFrame == #
govt_debt = pd.read_excel('gd.xls', sheetname='Data', skiprows=3, index_col=1)

# == Take desired values and plot == #
govt_debt = govt_debt.transpose()
govt_debt = govt_debt[['AUS', 'USA']]
govt_debt = govt_debt[38:]
govt_debt.plot(lw=2)
plt.show()

(The file is pandas/wb_download.py, and can be downloaded here)

The figure it produces looks as follows

![Graph showing government debt data for AUS and USA over time]

4.1.5 Exercises

Exercise 1

Write a program to calculate the percentage price change over 2013 for the following shares

ticker_list = {'INTC': 'Intel',
              'MSFT': 'Microsoft',
              'IBM': 'IBM',
              'BHP': 'BHP',
              'TM': 'Toyota',
              'AAPL': 'Apple',
              'AMZN': 'Amazon',
              'BA': 'Boeing',
              'QCOM': 'Qualcomm',}
A dataset of daily closing prices for the above firms can be found in pandas/data/ticker_data.csv, and can be downloaded here.

Plot the result as a bar graph like follows:

4.1.6 Solutions

Exercise 1

ticker = pd.read_csv('https://github.com/QuantEcon/QuantEcon.lectures.code/raw/master/pandas/data/ticker_data.csv')
ticker.set_index('Date', inplace=True)

ticker_list = {'INTC': 'Intel',
               'MSFT': 'Microsoft',
               'IBM': 'IBM',
               'BHP': 'BHP',
               'TM': 'Toyota',
               'KO': 'Coca-Cola',
               'GOOG': 'Google',
               'SNE': 'Sony',
               'PTR': 'PetroChina'}
AAPL': 'Apple',
'AMZN': 'Amazon',
'BA': 'Boeing',
'QCOM': 'Qualcomm',
'KO': 'Coca-Cola',
'GOOG': 'Google',
'SNE': 'Sony',
'PTR': 'PetroChina'}

price_change = pd.Series()

for tick in ticker_list:
    change = 100 * (ticker.loc[ticker.index[-1], tick] - ticker.loc[ticker.index[0], tick]) / ticker.loc[ticker.index[0], tick]
    name = ticker_list[tick]
    price_change[name] = change

price_change.sort_values(inplace=True)

fig, ax = plt.subplots(figsize=(10, 8))
price_change.plot(kind='bar', ax=ax)
plt.show()
4.2 Pandas for Panel Data

Contents

- Pandas for Panel Data
  - Overview
  - Slicing and reshaping data
  - Merging dataframes and filling NaNs
  - Grouping and summarizing data
  - Final Remarks

4.2. Pandas for Panel Data
4.2.1 Overview

In an earlier lecture on pandas we looked at working with simple data sets. Econometricians often need to work with more complex data sets, such as panels. Common tasks include

- Importing data, cleaning it and reshaping it across several axes
- Selecting a time series or cross-section from a panel
- Grouping and summarizing data

**pandas** (derived from panel and data) contains powerful and easy-to-use tools for solving exactly these kinds of problems.

In what follows, we will use a panel data set of real minimum wages from the OECD to create:

- summary statistics over multiple dimensions of our data
- a time series of the average minimum wage of countries in the dataset
- kernel density estimates of wages by continent

We will begin by reading in our long format panel data from a csv file and reshaping the resulting DataFrame with **pivot_table** to build a MultiIndex.

Additional detail will be added to our DataFrame using pandas **merge** function, and data will be summarized with the **groupby** function.

Most of this lecture was created by Natasha Watkins.

4.2.2 Slicing and reshaping data

We will read in a dataset from the OECD of real minimum wages in 32 countries and assign it to **realwage**.

The dataset pandas_panel/realwage.csv can be downloaded here.

Make sure the file is in your current working directory.

```python
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('https://github.com/QuantEcon/QuantEcon.lectures.code/raw/master/pandas_panel/realwage.csv')
```
Lets have a look at what we've got to work with

```python
realwage.head()  # Show first 5 rows
```

The data is currently in long format, which is difficult to analyse 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.

```python
realwage = realwage.pivot_table(values='value',
                                index='Time',
                                columns=['Country', 'Series', 'Pay period'])
realwage.head()
```

To more easily filter our time series data later on, we will convert the index into a `DateTimeIndex`.

```python
realwage.index = pd.to_datetime(realwage.index)
type(realwage.index)
```
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

```python
type(realwage.columns)
```

```python
pandas.core.indexes.multi.MultiIndex
```

```python
realwage.columns.names
```

```python
FrozenList(['Country', 'Series', 'Pay period'])
```

Like before, we can select the country (the top level of our `MultiIndex`)

```python
realwage['United States'].head()
```

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)

```python
realwage.stack().head()
```
We can also pass in an argument to select the level we would like to stack:

```
realwage.stack(level='Country').head()
```

Using a `DatetimeIndex` makes it easy to select a particular time period:

Selecting one year and stacking the two lower levels of the `MultiIndex` creates a cross-section of our panel data:

```
realwage['2015'].stack(level=(1, 2)).transpose().head()
```
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):

```python
realwage_f = realwage.xs(('Hourly', 'In 2015 constant prices at 2015 USD exchange rates'), level=('Pay period', 'Series'), axis=1)
realwage_f.head()
```

![DataFrame head](image)

5 rows x 32 columns

### 4.2.3 Merging dataframes and filling NaNs

Similar to relational databases like SQL, pandas has built in methods to merge datasets together.

Using country information from WorldData.info, well add the continent of each country to realwage_f with the `merge` function.

The csv file can be found in pandas_panel/countries.csv, and can be downloaded here:

```python
worlddata = pd.read_csv('https://github.com/QuantEcon/QuantEcon.lectures.code/raw/master/pandas_panel/countries.csv', sep=';')
worlddata.head()
```

![DataFrame head](image)

5 rows x 17 columns
First well select just the country and continent variables from `worlddata` and rename the column to `Country`:

```python
worlddata = worlddata[['Country (en)', 'Continent']
worlddata = worlddata.rename(columns={'Country (en)': 'Country'})
worlddata.head()
```

<table>
<thead>
<tr>
<th>Country</th>
<th>Continent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afghanistan</td>
<td>Asia</td>
</tr>
<tr>
<td>Egypt</td>
<td>Africa</td>
</tr>
<tr>
<td>Åland Islands</td>
<td>Europe</td>
</tr>
<tr>
<td>Albania</td>
<td>Europe</td>
</tr>
<tr>
<td>Algeria</td>
<td>Africa</td>
</tr>
</tbody>
</table>

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:

```python
realwage_f.transpose().head()
```

<table>
<thead>
<tr>
<th>Time</th>
<th>2006-01-01 00:00:00</th>
<th>2007-01-01 00:00:00</th>
<th>2008-01-01 00:00:00</th>
<th>...</th>
<th>2014-01-01 00:00:00</th>
<th>2015-01-01 00:00:00</th>
<th>2016-01-01 00:00:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Australia</td>
<td>12.06</td>
<td>12.46</td>
<td>12.24</td>
<td>...</td>
<td>12.07</td>
<td>12.83</td>
<td>12.98</td>
</tr>
<tr>
<td>Belgium</td>
<td>9.70</td>
<td>9.82</td>
<td>9.87</td>
<td>...</td>
<td>10.01</td>
<td>9.95</td>
<td>9.76</td>
</tr>
<tr>
<td>Brazil</td>
<td>0.87</td>
<td>0.92</td>
<td>0.96</td>
<td>...</td>
<td>1.21</td>
<td>1.21</td>
<td>1.24</td>
</tr>
<tr>
<td>Canada</td>
<td>6.99</td>
<td>6.96</td>
<td>7.24</td>
<td>...</td>
<td>8.22</td>
<td>8.35</td>
<td>8.48</td>
</tr>
<tr>
<td>Chile</td>
<td>1.42</td>
<td>1.45</td>
<td>1.44</td>
<td>...</td>
<td>1.76</td>
<td>1.81</td>
<td>1.91</td>
</tr>
</tbody>
</table>

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.

![Venn Diagram](image)

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'`.

```python
merged = pd.merge(realwage_f.transpose(), worlddata,
                  how='left', left_index=True, right_on='Country')
merged.head()
```

<table>
<thead>
<tr>
<th></th>
<th>2006-01-01 00:00:00</th>
<th>2007-01-01 00:00:00</th>
<th>2008-01-01 00:00:00</th>
<th>...</th>
<th>2016-01-01 00:00:00</th>
<th>Country</th>
<th>Continent</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>12.06</td>
<td>12.46</td>
<td>12.24</td>
<td>...</td>
<td>12.98</td>
<td>Australia</td>
<td>Australia</td>
</tr>
<tr>
<td>23</td>
<td>9.70</td>
<td>9.82</td>
<td>9.87</td>
<td>...</td>
<td>9.76</td>
<td>Belgium</td>
<td>Europe</td>
</tr>
<tr>
<td>32</td>
<td>0.87</td>
<td>0.92</td>
<td>0.90</td>
<td>...</td>
<td>1.24</td>
<td>Brazil</td>
<td>South America</td>
</tr>
<tr>
<td>100</td>
<td>0.89</td>
<td>0.96</td>
<td>7.24</td>
<td>...</td>
<td>8.48</td>
<td>Canada</td>
<td>North America</td>
</tr>
<tr>
<td>38</td>
<td>1.42</td>
<td>1.45</td>
<td>1.44</td>
<td>...</td>
<td>1.91</td>
<td>Chile</td>
<td>South America</td>
</tr>
</tbody>
</table>

5 rows × 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.
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.

```python
missing_continents = {'Korea': 'Asia',
                     'Russian Federation': 'Europe',
                     'Slovak Republic': 'Europe'}
merged['Country'].map(missing_continents)
```
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.

```python
merged['Continent'] = merged['Continent'].fillna(merged['Country'].map(missing_continents))
```

# Check for whether continents were correctly mapped
merged[merged['Country'] == 'Korea']

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.

```python
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.

```python
merged = merged.set_index(['Continent', 'Country']).sort_index()
merged.head()
```
While merging, we lost our `DatetimeIndex`, as we merged columns that were not in datetime format.

```python
temporal_columns = merged.index
```

Now that we have set the merged columns as the index, we can recreate a `DatetimeIndex` using `.to_datetime()`.

```python
temporal_columns = pd.to_datetime(merged.columns)
temporal_columns = temporal_columns.rename('Time')
```

The `DatetimeIndex` tends to work more smoothly in the row axis, so we will go ahead and transpose.

```python
merged = merged.transpose()
merged.head()
```
4.2.4 Grouping and summarizing data

Grouping and summarizing data can be particularly useful for understanding large panel datasets.

A simple way to summarize data is to call an aggregation method on the dataframe, such as \texttt{.mean()} or \texttt{.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).

\begin{verbatim}
merged.mean().head(10)
\end{verbatim}

Using this series, we can plot the average real minimum wage over the past decade for each country in our data set.

\begin{verbatim}
import matplotlib.pyplot as plt
import matplotlib
matplotlib.style.use('seaborn')
\end{verbatim}
Passing in `axis=1` to `.mean()` will aggregate over columns (giving the average minimum wage for all countries over time).
merged.mean(axis=1).head()

<table>
<thead>
<tr>
<th>Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006-01-01</td>
<td>4.69</td>
</tr>
<tr>
<td>2007-01-01</td>
<td>4.84</td>
</tr>
<tr>
<td>2008-01-01</td>
<td>4.90</td>
</tr>
<tr>
<td>2009-01-01</td>
<td>5.08</td>
</tr>
<tr>
<td>2010-01-01</td>
<td>5.11</td>
</tr>
</tbody>
</table>

dtype: float64

We can plot this time series as a line graph

```python
merged.mean(axis=1).plot()
plt.title('Average real minimum wage 2006 - 2016')
plt.ylabel('2015 USD')
plt.xlabel('Year')
plt.show()
```

![Average real minimum wage 2006 - 2016](image)

We can also specify a level of the `MultiIndex` (in the column axis) to aggregate over

```python
merged.mean(level='Continent', axis=1).head()
```
We can plot the average minimum wages in each continent as a time series:

```python
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

```python
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

```python
merged.stack().describe()
```
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:

```python
grouped = merged.groupby(level='Continent', axis=1)
grouped
```

< pandas.core.groupby.DataFrameGroupBy object at 0x7f2900efd160 >

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.

```python
grouped.size()
```

<table>
<thead>
<tr>
<th>Continent</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>America</td>
<td>7</td>
</tr>
<tr>
<td>Asia</td>
<td>4</td>
</tr>
</tbody>
</table>
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.

```python
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()
```
4.2.5 Final Remarks

This lecture has provided an introduction to some of pandas more advanced features, including multiindices, merging, grouping and plotting

Other tools that may be useful in panel data analysis include xarray, a python package that extends pandas to N-dimensional data structures

4.2.6 Exercises

Exercise 1

In these exercises you’ll work with a dataset of employment rates in Europe by age and sex from Eurostat

The dataset pandas_panel/employ.csv can be downloaded here

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

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

4.2.7 Solutions

Exercise 1

```python
employ = pd.read_csv('https://github.com/QuantEcon/QuantEcon.lectures.code/raw/master/pandas_panel/employ.csv')
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
employ.head()
```
This is a large dataset so it is useful to explore the levels and variables available.

```python
employ.columns.names
```

```
FrozenList(['UNIT', 'AGE', 'SEX', 'INDIC_EM', 'GEO'])
```

Variables within levels can be quickly retrieved with a loop.

```python
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',...]
```

5 rows × 1440 columns
Exercise 2

To easily filter by country, swap `GEO` to the top level and sort the `MultiIndex`

```python
df = df.copy()  # Assume df is a DataFrame

df['GEO'] = df.columns.get_level_values('GEO').unique().tolist()
countries = [x for x in df['GEO'].unique() if not x.startswith('Eu')]
df = df[countries].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`

```python
df['GEO'] = df['GEO'].unique().tolist()
countries = [x for x in df['GEO'].unique() if not x.startswith('Euro')]
df = df[countries].sort_index(axis=1)
```

Select only percentage employed in the active population from the dataframe

```python
df = df.xs(('Percentage of total population', 'Active population'), level=('UNIT', 'INDIC_EM'), axis=1)
df.head()
```
Drop the Total value before creating the grouped boxplot

```python
employ_f = employ_f.drop('Total', level='SEX', axis=1)

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()
```
4.3 Linear Regression in Python

**Contents**

- Linear Regression in Python
  - Overview
  - Simple Linear Regression
  - Extending the Linear Regression Model
  - Endogeneity
  - Summary
  - Exercises
  - Solutions
4.3.1 Overview

Linear regression is a standard tool for analyzing the relationship between two or more variables. In this lecture, we will use the Python package `statsmodels` to estimate, interpret, and visualize linear regression models. Along the way, we will discuss a variety of topics, including:

- simple and multivariate linear regression
- visualization
- endogeneity and omitted variable bias
- two-stage least squares

As an example, we will replicate results from Acemoglu, Johnson and Robinson's seminal paper [AJR01].

You can download a copy here.

In the paper, the authors emphasize the importance of institutions in economic development.

The main contribution is the use of settler mortality rates as a source of exogenous variation in institutional differences. Such variation is needed to determine whether it is institutions that give rise to greater economic growth, rather than the other way around.

Prerequisites

This lecture assumes you are familiar with basic econometrics.

For an introductory text covering these topics, see, for example, [Woo15].

Comments

This lecture is coauthored with Natasha Watkins.

4.3.2 Simple Linear Regression

[AJR01] wish to determine whether or not differences in institutions can help to explain observed economic outcomes.

How do we measure institutional differences and economic outcomes?

In this paper,

- economic outcomes are proxied by log GDP per capita in 1995, adjusted for exchange rates
- institutional differences are proxied by an index of protection against expropriation on average over 1985-95, constructed by the Political Risk Services Group.
These variables and other data used in the paper are available for download on Daron Acemoglus webpage. We will use pandas .read_stata() function to read in data contained in the .dta files to dataframes.

```python
import pandas as pd
df1 = pd.read_stata('https://github.com/QuantEcon/QuantEcon.lectures.code/raw/master/ols/maketable1.dta')
df1.head()
```

```
shortnam euro1990 excolony avexpr logpgp95 cons1 cons90 democ90a cons00a extmrt4 logem4 loghypj baseco
0 AFG  0.000000  1.0   NaN   NaN   1.0   2.0   1.0   1.0   93.699997  4.540998   NaN  NaN
1 AGO  8.000000   1.0  5.383636  7.770645  3.0   3.0   0.0   1.0  280.000000  5.634789 -3.411248   1.0
2 ARG  0.000000   1.0  7.191818  9.804219  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
3 ARM  60.000004  1.0  6.383364  9.133459  1.0   6.0   3.0   3.0  68.900002  4.223556 -0.072274   1.0
4 ARM  0.000000  1.0   NaN  7.682482  NaN  NaN  NaN  NaN  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.

```python
import matplotlib.pyplot as plt
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:

- $\beta_0$ is the intercept of the linear trend line on the y-axis
- $\beta_1$ is the slope of the linear trend line, representing the marginal effect of protection against risk on log GDP per capita
- $u_i$ is a random error term (deviations of observations from the linear trend due to factors not included in the model)

Visually, this linear model involves choosing a straight line that best fits the data, as in the following plot (Figure 2 in [AJR01])

```python
import numpy as np

# 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
plt.scatter(X, y, marker='')

for i, label in enumerate(labels):
    plt.annotate(label, (X.iloc[i], y.iloc[i]))

# Fit a linear trend line
plt.plot(np.unique(X),
         np.poly1d(np.polyfit(X, y, 1))(np.unique(X)),
         color='black')

plt.xlim([3.3, 10.5])
plt.ylim([4, 10.5])
plt.xlabel('Average Expropriation Risk 1985-95')
plt.ylabel('Log GDP per capita, PPP, 1995')
plt.title('Figure 2: OLS relationship between expropriation risk and income')
plt.show()
```
The most common technique to estimate the parameters ($\beta$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.

$$
\min_{\beta} \sum_{i=1}^{N} \hat{u}_i^2
$$

where $\hat{u}_i$ is the difference between the observation and the predicted value of the dependent variable

To estimate the constant term $\beta_0$, we need to add a column of 1s to our dataset (consider the equation if $\beta_0$ was replaced with $\beta_0 x_i$ and $x_i = 1$)

```python
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

```python
import statsmodels.api as sm
```
reg1 = sm.OLS(endog=df1['logpgp95'], exog=df1[['const', 'avexpr']], missing='drop')
type(reg1)

So far we have simply constructed our model.

We need to use `.fit()` to obtain parameter estimates $\hat{\beta}_0$ and $\hat{\beta}_1$.

```
results = reg1.fit()
type(results)
```

We now have the fitted regression model stored in `results`.

To view the OLS regression results, we can call the `.summary()` method.

```
print(results.summary())
```

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

$$\text{logpgp}_{95i} = 4.63 + 0.53 \text{avexpr}_i$$

This equation describes the line that best fits our data, as shown in Figure 2

We can use this equation to predict the level of log GDP per capita for a value of the index of expropriation protection

For example, for a country with an index value of 7.07 (the average for the dataset), we find that their predicted level of log GDP per capita in 1995 is 8.38

```
mean_expr = np.mean(df1_subset['avexpr'])
mean_expr
6.515625

predicted_logpdp95 = 4.63 + 0.53 * 7.07
predicted_logpdp95
8.3771
```

An easier (and more accurate) way to obtain this result is to use `predict()` and set `constant = 1` and `avexpr_i = mean_expr`

```
results.predict(exog=[1, mean_expr])
array([ 8.09156367])
```

We can obtain an array of predicted $\text{logpgp}_{95i}$ for every value of $\text{avexpr}_i$ in our dataset by calling `predict()` on our results

Plotting the predicted values against $\text{avexpr}_i$ shows that the predicted values lie along the linear line that we fitted above

The observed values of $\text{logpgp}_{95i}$ are also plotted for comparison purposes

```
# Drop missing observations from whole sample
df1_plot = df1.dropna(subset=['logpgp95', 'avexpr'])
```
# Plot predicted values

plt.scatter(df1_plot['avexpr'], results.predict(), alpha=0.5, label='predicted')

# Plot observed values

plt.scatter(df1_plot['avexpr'], df1_plot['logpgp95'], alpha=0.5, label='observed')

plt.legend()
plt.title('OLS predicted values')
plt.xlabel('avexpr')
plt.ylabel('logpgp95')
plt.show()

### 4.3.3 Extending the Linear Regression Model

So far we have only accounted for institutions affecting economic performance - almost certainly there are numerous other factors affecting GDP that are not included in our model
Leaving out variables that affect \( \log pgp_{95} \), 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 \( \log pgp_{95} \). [AJR01] consider other factors such as:

- the effect of climate on economic outcomes; latitude is used to proxy this
- differences that affect both economic performance and institutions, eg. cultural, historical, etc.; controlled for with the use of continent dummies

Let's estimate some of the extended models considered in the paper (Table 2) using data from `maketable2.dta`

```python
import pandas as pd
from statsmodels.iolib.summary2 import summary_col

# Download the data
df2 = pd.read_stata('https://github.com/QuantEcon/QuantEcon.lectures.code/raw/master/ols/maketable2.dta')

# Add constant term to dataset
df2['const'] = 1

# Create lists of variables to be used in each regression
X1 = ['const', 'avexpr']
X2 = ['const', 'avexpr', 'lat_abst']
X3 = ['const', 'avexpr', 'lat_abst', 'asia', 'africa', 'other']

# Estimate an OLS regression for each set of variables
reg1 = sm.OLS(df2['logpgp95'], df2[X1], missing='drop').fit()
reg2 = sm.OLS(df2['logpgp95'], df2[X2], missing='drop').fit()
reg3 = sm.OLS(df2['logpgp95'], df2[X3], missing='drop').fit()

# Add constant term to dataset

def summary_col(results=[reg1, reg2, reg3],
                 float_format='%0.2f',
                 stars=True,
                 model_names=['Model 1', 'Model 3', 'Model 4'],
                 info_dict=None,
                 regressor_order=['const', 'avexpr', 'lat_abst', 'asia', 'africa'])

        results_table.add_title('Table 2 - OLS Regressions')
```

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)
print(results_table)

<table>
<thead>
<tr>
<th>Table 2 - OLS Regressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>Model 1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Model 3</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Model 4</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01

4.3.4 Endogeneity

As [AJR01] discuss, the OLS models likely suffer from **endogeneity** issues, resulting in biased and inconsistent model estimates. Namely, there is likely a two-way relationship between institutions and economic outcomes:

- richer countries may be able to afford or prefer better institutions
- variables that affect income may also be correlated with institutional differences
- the construction of the index may be biased; analysts may be biased towards seeing countries with higher income having better institutions

To deal with endogeneity, we can use **two-stage least squares (2SLS) regression**, which is an extension of OLS regression.

This method requires replacing the endogenous variable $avexpr_i$ with a variable that is:

1. correlated with $avexpr_i$
2. not correlated with the error term (i.e. it should not directly affect the dependent variable, otherwise it would be correlated with $u_i$ due to omitted variable bias)

The new set of regressors is called an **instrument**, which aims to remove endogeneity in our proxy of institutional differences.
The main contribution of [AJR01] is the use of settler mortality rates to instrument for institutional differences. They hypothesize that higher mortality rates of colonizers led to the establishment of institutions that were more extractive in nature (less protection against expropriation), and these institutions still persist today.

Using a scatterplot (Figure 3 in [AJR01]), we can see protection against expropriation is negatively correlated with settler mortality rates, coinciding with the authors hypothesis and satisfying the first condition of a valid instrument.

```python
# 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
plt.scatter(X, y, marker='')
for i, label in enumerate(labels):
    plt.annotate(label, (X.iloc[i], y.iloc[i]))

# Fit a linear trend line
plt.plot(np.unique(X),
         np.poly1d(np.polyfit(X, y, 1))(np.unique(X)),
         color='black')

plt.xlim([1.8, 8.4])
plt.ylim([3.3, 10.4])
plt.xlabel('Log of Settler Mortality')
plt.ylabel('Average Expropriation Risk 1985-95')
plt.title('Figure 3: First-stage relationship between settler mortality and expropriation risk')
plt.show()
```
The second condition may not be satisfied if settler mortality rates in the 17th to 19th centuries have a direct
effect on current GDP (in addition to their indirect effect through institutions)

For example, settler mortality rates may be related to the current disease environment in a country, which
could affect current economic performance

[AJR01] argue this is unlikely because:

- The majority of settler deaths were due to malaria and yellow fever, and had 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 \(logem_i\)
Therefore, we will estimate the first-stage regression as

\[ \text{avexpr}_i = \delta_0 + \delta_1 \text{logem}_4 + v_i \]

The data we need to estimate this equation is located in `maketable4.dta` (only complete data, indicated by `baseco = 1`, is used for estimation)

```python
# Import and select the data
df4 = pd.read_stata('https://github.com/QuantEcon/QuantEcon.lectures.code/raw/master/ols/maketable4.dta')
df4 = df4[df4['baseco'] == 1]

# Add a constant variable
df4['const'] = 1

# Fit the first stage regression and print summary
results_fs = sm.OLS(df4['avexpr'],
                    df4[['const', 'logem4']],
                    missing='drop').fit()
print(results_fs.summary())
```

Omitting an important point in the documentation...

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 `\hat{avexpr}_i` in the original linear model.
Our second stage regression is thus

\[ \log pgp_{95} = \beta_0 + \beta_1 \text{avexpr}_i + u_i \]

```python
df4['predicted_avexpr'] = results_fs.predict()
results_ss = sm.OLS(df4['logpgp95'],
                    df4[['const', 'predicted_avexpr']]).fit()
print(results_ss.summary())
```

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`.

To install this package, you will need to run `pip install linearmodels` in your command line.

300 Chapter 4. Data and Empirics
from linearmodels.iv import IV2SLS

Note that when using IV2SLS, the exogenous and instrument variables are split up in the function arguments (whereas before the instrument included exogenous variables)

```python
iv = IV2SLS(dependent=df4['logpgp95'],
            exog=df4['const'],
            endog=df4['avexpr'],
            instruments=df4['logem4']).fit(cov_type='unadjusted')

print(iv.summary)
```

<table>
<thead>
<tr>
<th>IV-2SLS Estimation Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dep. Variable: logpgp95 R-squared: 0.1870</td>
</tr>
<tr>
<td>Estimator: IV-2SLS Adj. R-squared: 0.1739</td>
</tr>
<tr>
<td>No. Observations: 64 F-statistic: 37.568</td>
</tr>
<tr>
<td>Date: Mon, Jul 17 2017 P-value (F-stat) 0.0000</td>
</tr>
<tr>
<td>Time: 18:41:29 Distribution: chi2(1)</td>
</tr>
<tr>
<td>Cov. Estimator: unadjusted</td>
</tr>
</tbody>
</table>

Parameter Estimates

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Std. Err.</th>
<th>T-stat</th>
<th>P-value</th>
<th>Lower CI</th>
<th>Upper CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>1.9097</td>
<td>1.0106</td>
<td>1.8897</td>
<td>0.0588</td>
<td>-0.0710</td>
</tr>
<tr>
<td>avexpr</td>
<td>0.9443</td>
<td>0.1541</td>
<td>6.1293</td>
<td>0.0000</td>
<td>0.6423</td>
</tr>
</tbody>
</table>

Given that we now have consistent and unbiased estimates, we can infer from the model we have estimated that institutional differences (stemming from institutions set up during colonization) can help to explain differences in income levels across countries today.

[AJR01] use a marginal effect of 0.94 to calculate that the difference in the index between Chile and Nigeria (ie. institutional quality) implies up to a 7-fold difference in income, emphasizing the significance of institutions in economic development.

4.3.5 Summary

We have demonstrated basic OLS and 2SLS regression in statsmodels and linearmodels.

If you are familiar with R, you may want use the formula interface to statsmodels, or consider using r2py to call R from within Python.

4.3. Linear Regression in Python
4.3.6 Exercises

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, $u_i$

$$H_0 : \text{Cov}(avexpr_i, u_i) = 0 \quad (\text{no endogeneity})$$

$$H_1 : \text{Cov}(avexpr_i, u_i) \neq 0 \quad (\text{endogeneity})$$

This test is run in two stages:

First, we regress $avexpr_i$ on the instrument, $logem4_i$:

$$avexpr_i = \pi_0 + \pi_1 logem4_i + v_i$$

Second, we retrieve the residuals $\hat{v}_i$ and include them in the original equation:

$$logpgp95_i = \beta_0 + \beta_1 avexpr_i + \alpha \hat{v}_i + u_i$$

If $\alpha$ 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.

Exercise 2

The OLS parameter $\beta$ 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 $\beta$, we want to minimise the sum of squared residuals:

$$\min_{\beta} \hat{u}'\hat{u}$$

Rearranging the first equation and substituting into the second equation, we can write:

$$\min_{\beta} (Y - X\hat{\beta})'(Y - X\hat{\beta})$$

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.
4.3.7 Solutions

Exercise 1

```python
# Load in data
df4 = pd.read_stata('https://github.com/QuantEcon/QuantEcon.lectures.code/raw/master/ols/maketable4.dta')

# Add a constant term
df4['const'] = 1

# Estimate the first stage regression
reg1 = sm.OLS(endog=df4['avexpr'],
              exog=df4[['const', 'logem4']],
              missing='drop').fit()

# Retrieve the residuals
df4['resid'] = reg1.resid

# Estimate the second stage residuals
reg2 = sm.OLS(endog=df4['logpgp95'],
              exog=df4[['const', 'avexpr', 'resid']],
              missing='drop').fit()

print(reg2.summary())
```

### OLS Regression Results

<table>
<thead>
<tr>
<th>Dep. Variable:</th>
<th>logpgp95</th>
<th>R-squared:</th>
<th>0.689</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model:</td>
<td>OLS</td>
<td>Adj. R-squared:</td>
<td>0.679</td>
</tr>
<tr>
<td>Method:</td>
<td>Least Squares F-statistic:</td>
<td>74.05</td>
<td></td>
</tr>
<tr>
<td>Date:</td>
<td>Mon, 17 Jul 2017</td>
<td>Prob (F-statistic):</td>
<td>1.07e-17</td>
</tr>
<tr>
<td>Time:</td>
<td>18:41:29</td>
<td>Log-Likelihood:</td>
<td>-62.031</td>
</tr>
<tr>
<td>No. Observations:</td>
<td>70</td>
<td>AIC:</td>
<td>130.1</td>
</tr>
<tr>
<td>Df Residuals:</td>
<td>67</td>
<td>BIC:</td>
<td>136.8</td>
</tr>
<tr>
<td>Df Model:</td>
<td>2</td>
<td>Cond. No.</td>
<td>53.8</td>
</tr>
</tbody>
</table>

| 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.

**Exercise 2**

```python
# Load in data
df1 = pd.read_stata('https://github.com/QuantEcon/QuantEcon.lectures.code/raw/ master/ols/maketable1.dta')
df1 = df1.dropna(subset=['logpgp95', 'avexpr'])

# Add a constant term
df1['const'] = 1

# Define the X and y variables
y = np.asarray(df1['logpgp95'])
X = np.asarray(df1[['const', 'avexpr']])

# Compute $\hat{\beta}_\text{hat}$
beta_hat = np.linalg.solve(X.T @ X, X.T @ y)

# Print out the results from the 2 x 1 vector $\hat{\beta}_\text{hat}$
print(f'\beta_0 = ({beta_hat[0]:.2})')
print(f'\beta_1 = ({beta_hat[1]:.2})')

beta_0 = 4.6
beta_1 = 0.53
```

It is also possible to use `np.linalg.inv(X.T @ X) @ X.T @ y` to solve for $\beta$, however `.solve()` is preferred as it involves fewer computations.

### 4.4 Maximum Likelihood Estimation

**Contents**

- **Maximum Likelihood Estimation**
  - Overview
  - Set Up and Assumptions
  - Conditional Distributions
  - Maximum Likelihood Estimation
    - MLE with Numerical Methods
    - Maximum Likelihood Estimation with `statsmodels`
4.4.1 Overview

In a previous lecture we estimated the relationship between dependent and explanatory variables using linear regression. But what if a linear relationship is not an appropriate assumption for our model? One widely used alternative is maximum likelihood estimation, which involves specifying a class of distributions, indexed by unknown parameters, and then using the data to pin down these parameter values. The benefit relative to linear regression is that it allows more flexibility in the probabilistic relationships between variables.

Here we illustrate maximum likelihood by replicating Daniel Treismans (2016) paper, Russias 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.

Prerequisites

We assume familiarity with basic probability and multivariate calculus.

Comments

This lecture is coauthored with Natasha Watkins.

4.4.2 Set Up and Assumptions

Let's consider the steps we need to go through in maximum likelihood estimation and how they pertain to this study.

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 $\sigma \in (0, \infty)$

Well 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**

**Counting Billionaires**

Treisman [*Tre16*] is interested in estimating the number of billionaires in different countries

The number of billionaires is integer valued

Hence we consider distributions that take values only in the nonnegative integers

(This is one reason least squares regression is not the best tool for the present problem, since the dependent variable in linear regression is not restricted to integer values)

One integer distribution is the **Poisson distribution**, the probability mass function (pmf) of which is

$$f(y) = \frac{\mu^y}{y!} e^{-\mu}, \quad y = 0, 1, 2, \ldots, \infty$$

We can plot the Poisson distribution over $y$ for different values of $\mu$ as follows

```python
from numpy import exp
from scipy.special import factorial
import matplotlib.pyplot as plt

poisson_pmf = lambda y, mu: mu**y / factorial(y) * exp(-mu)
y_values = range(0, 25)

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

for mu in [1, 5, 10]:
    distribution = []
    for y_i in y_values:
        distribution.append(poisson_pmf(y_i, mu))
    ax.plot(y_values, distribution, label='$\mu$={}'.format(mu), alpha=0.5, marker='o', markersize=8)
ax.grid()
ax.set_xlabel('$y$', fontsize=14)
ax.set_ylabel('$f(y \mid \mu)$', fontsize=14)
ax.axis(xmin=0, ymin=0)
ax.legend(fontsize=14)
plt.show()
```
Notice that the Poisson distribution begins to resemble a normal distribution as the mean of $y$ increases.

Let's have a look at the distribution of the data we'll be working with in this lecture.

Treisman's main source of data is *Forbes* annual rankings of billionaires and their estimated net worth.

The dataset `mle.fp.dta` can be downloaded from [here](https://github.com/QuantEcon/QuantEcon.lectures.code/raw/master/mle/fp.dta) or its [AER page](https://www.aer.ucalgary.ca/).

```python
import pandas as pd
pd.options.display.max_columns = 10

# Load in data and view
df = pd.read_stata('https://github.com/QuantEcon/QuantEcon.lectures.code/raw/master/mle/fp.dta')
df.head()
```

<table>
<thead>
<tr>
<th>country</th>
<th>ccode</th>
<th>year</th>
<th>cyear</th>
<th>numbil</th>
<th>...</th>
<th>topint08</th>
<th>rintr</th>
<th>noyrs</th>
<th>roflaw</th>
<th>nrrents</th>
</tr>
</thead>
<tbody>
<tr>
<td>United States</td>
<td>2.0</td>
<td>1990.0</td>
<td>21990.0</td>
<td>NaN</td>
<td>...</td>
<td>39.799999</td>
<td>4.988405</td>
<td>20.0</td>
<td>1.61</td>
<td>NaN</td>
</tr>
<tr>
<td>United States</td>
<td>2.0</td>
<td>1991.0</td>
<td>21991.0</td>
<td>NaN</td>
<td>...</td>
<td>39.799999</td>
<td>4.988405</td>
<td>20.0</td>
<td>1.61</td>
<td>NaN</td>
</tr>
<tr>
<td>United States</td>
<td>2.0</td>
<td>1992.0</td>
<td>21992.0</td>
<td>NaN</td>
<td>...</td>
<td>39.799999</td>
<td>4.988405</td>
<td>20.0</td>
<td>1.61</td>
<td>NaN</td>
</tr>
<tr>
<td>United States</td>
<td>2.0</td>
<td>1993.0</td>
<td>21993.0</td>
<td>NaN</td>
<td>...</td>
<td>39.799999</td>
<td>4.988405</td>
<td>20.0</td>
<td>1.61</td>
<td>NaN</td>
</tr>
<tr>
<td>United States</td>
<td>2.0</td>
<td>1994.0</td>
<td>21994.0</td>
<td>NaN</td>
<td>...</td>
<td>39.799999</td>
<td>4.988405</td>
<td>20.0</td>
<td>1.61</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Using a histogram, we can view the distribution of the number of billionaires per country, `numbilo`, in 2008 (the United States is dropped for plotting purposes)

```python
numbilo_2008 = df[(df['year'] == 2008) & (df['country'] != 'United States')].loc[:, 'numbilo']
plt.subplots(figsize=(12, 8))
plt.hist(numbilo_2008, bins=30)
plt.xlim(xmin=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 \( \mu \) and some outliers)

### 4.4.3 Conditional Distributions

In Treismans 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 \( x_i \)

The standard formulation the so-called poisson regression model is as follows:
\[ f(y_i \mid x_i) = \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}; \quad y_i = 0, 1, 2, \ldots, \infty. \tag{4.1} \]

where \( \mu_i = \exp(x_i'\beta) = \exp(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_k x_{ik}) \)

To illustrate the idea that the distribution of \( y_i \) depends on \( x_i \), let's run a simple simulation.

We use our poisson_pmf function from above and arbitrary values for \( \beta \) and \( x_i \).

```python
import numpy as np

y_values = range(0, 20)

# Define a parameter vector with estimates
beta = np.array([0.26, 0.18, 0.25, -0.1, -0.22]).T

# 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:
    mu = exp(X @ beta)
    distribution = []
    for y_i in y_values:
        distribution.append(poisson_pmf(y_i, mu))
    ax.plot(y_values, distribution,
            label=f'$\mu_i=${mu:.1f}',
            marker='o',
            markersize=8,
            alpha=0.5)

ax.grid()
ax.legend()
ax.set_xlabel(r'$y \mid x_i$')
ax.set_ylabel(r'$f(y \mid x_i; \beta)$')
ax.axis(xmin=0, ymin=0)
plt.show()
```

4.4. Maximum Likelihood Estimation
We can see that the distribution of $y_i$ is conditional on $x_i$ ($\mu_i$ is no longer constant)

### 4.4.4 Maximum Likelihood Estimation

In our model for number of billionaires, the conditional distribution contains 4 ($k = 4$) parameters that we need to estimate.

We will label our entire parameter vector as $\beta$ 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 $\beta$.

Intuitively, we want to find the $\hat{\beta}$ that best fits our data.

First we need to construct the likelihood function $L(\beta)$, 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 $f(y_1, y_2) = f(y_1) \cdot f(y_2)$. 

310 Chapter 4. Data and Empirics
If \( y_i \) follows a Poisson distribution with \( \lambda = 7 \), we can visualize the joint pmf like so

```python
from mpl_toolkits.mplot3d import Axes3D
def plot_joint_poisson(\( \mu \), \( y_n \)):
    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, \( \mu \)) * poisson_pmf(Y, \( \mu \))

    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(\( \mu = 7 \), \( y_n = 20 \))
```

Similarly, the joint pmf of our data (which is distributed as a conditional Poisson distribution) can be written.
as
\[
f(y_1, y_2, \ldots, y_n \mid x_1, x_2, \ldots, x_n; \beta) = \prod_{i=1}^{n} \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}
\]
y_i is conditional on both the values of x_i and the parameters \(\beta\)

The likelihood function is the same as the joint pmf, but treats the parameter \(\beta\) as a random variable and takes the observations \((y_i, x_i)\) as given

\[
\mathcal{L}(\beta \mid y_1, y_2, \ldots, y_n ; x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}
\]

\[
= f(y_1, y_2, \ldots, y_n \mid x_1, x_2, \ldots, x_n; \beta)
\]

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

\[
\log \mathcal{L}(\beta) = \log \left( f(y_1; \beta) \cdot f(y_2; \beta) \cdot \ldots \cdot f(y_n; \beta) \right)
\]

\[
= \sum_{i=1}^{n} \log f(y_i; \beta)
\]

\[
= \sum_{i=1}^{n} \log \left( \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i} \right)
\]

\[
= \sum_{i=1}^{n} y_i \log \mu_i - \sum_{i=1}^{n} \mu_i - \sum_{i=1}^{n} \log y_i!
\]

The MLE of the Poisson to the Poisson for \(\hat{\beta}\) can be obtained by solving

\[
\max_{\beta} \left( \sum_{i=1}^{n} y_i \log \mu_i - \sum_{i=1}^{n} \mu_i - \sum_{i=1}^{n} \log y_i! \right)
\]

However, no analytical solution exists to the above problem – to find the MLE we need to use numerical methods

### 4.4.5 MLE with Numerical Methods

Many distributions do not have nice, analytical solutions and therefore require numerical methods to solve for parameter estimates

One such numerical method is the Newton-Raphson algorithm
Our goal is 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

\[
\log L(\beta) = -(\beta - 10)^2 - 10
\]

```python
beta = np.linspace(1, 20)
logL = -(beta - 10) ** 2 - 10
dlogL = -2 * beta + 20

fig, (ax1, ax2) = plt.subplots(2, sharex=True, figsize=(12, 8))
ax1.plot(beta, logL, lw=2)
ax2.plot(beta, dlogL, lw=2)

ax1.set_ylabel(r'$\log \mathcal{L}(\beta)$', rotation=0, labelpad=35, fontsize=15)
ax2.set_ylabel(r'$\frac{d \log \mathcal{L}(\beta)}{d \beta}$', 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 \( \frac{d \log L(\beta)}{d\beta} = 0 \) (the bottom plot).

Therefore, the likelihood is maximized when \( \beta = 10 \).

We can also ensure that this value is a \textit{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, \( \beta_0 \) (the OLS parameter estimates might be a reasonable guess), then

1. Use the updating rule to iterate the algorithm

   \[ \beta_{(k+1)} = \beta_{(k)} - \frac{G(\beta_{(k)})}{H(\beta_{(k)})} \]

   where:

   \[ G(\beta_{(k)}) = \frac{d \log L(\beta_{(k)})}{d\beta_{(k)}} \]

   \[ H(\beta_{(k)}) = \frac{d^2 \log L(\beta_{(k)})}{d\beta_{(k)}^2} \]

2. Check whether \( \beta_{(k+1)} - \beta_{(k)} < tol \)

   • If true, then stop iterating and set \( \hat{\beta} = \beta_{(k+1)} \)
• If false, then update $\beta_{(k+1)}$.

As can be seen from the updating equation, $\beta_{(k+1)} = \beta_{(k)}$ only when $G(\beta_{(k)}) = 0$ ie. where the first derivative is equal to 0.

(In practice, we stop iterating when the difference is below a small tolerance threshold)

Lets have a go at implementing the Newton-Raphson algorithm.

First, well create a class called PoissonRegression so we can easily recompute the values of the log likelihood, gradient and Hessian for every iteration:

```python
class PoissonRegression:
    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 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):
        μ = self.μ()
        return ((self.y - μ) @ self.X).reshape(self.k, 1)

    def H(self):
        X = self.X
        μ = self.μ()
        return -(μ * X.T @ X)
```

Our function newton_raphson will take a PoissonRegression object that has an initial guess of the parameter vector $\beta_0$.

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:

```python
def newton_raphson(model, tol=1e-3, max_iter=1000, display=True):
    i = 0
    error = 100  # Initial error value
    # Print header of output
```

4.4. Maximum Likelihood Estimation 315
```python
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).T
        error = β_new - model.β
        model.β = β_new.flatten()

        # Print iterations
        if display:
            β_list = [f'{{t:.3}}' for t in list(model.β)]
            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)')

return model.β

Lets try out our algorithm with a small dataset of 5 observations and 3 variables in X

X = np.array([[1, 2, 5],
              [1, 1, 3],
              [1, 4, 2],
              [1, 5, 2],
              [1, 3, 1]])

y = np.array([1, 0, 1, 1, 0])

# Take a guess at initial βs
init_β = np.array([0.1, 0.1, 0.1])

# Create an object with Poisson model values
poi = PoissonRegression(y, X, β=init_β)

# Use newton_raphson to find the MLE
β_hat = newton_raphson(poi, display=True)
```

<table>
<thead>
<tr>
<th>Iteration_k</th>
<th>Log-likelihood</th>
<th>θ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-4.34476224</td>
<td>[-1.4890, 0.2650, 0.2440]</td>
</tr>
<tr>
<td>1</td>
<td>-3.5742413</td>
<td>[-3.3840, 0.5280, 0.4740]</td>
</tr>
<tr>
<td>2</td>
<td>-3.3995256</td>
<td>[-5.0640, 0.7820, 0.7020]</td>
</tr>
<tr>
<td>3</td>
<td>-3.37886465</td>
<td>[-5.9150, 0.9090, 0.8200]</td>
</tr>
<tr>
<td>4</td>
<td>-3.3783559</td>
<td>[-6.0740, 0.9330, 0.8430]</td>
</tr>
<tr>
<td>5</td>
<td>-3.37835551</td>
<td>[-6.0780, 0.9330, 0.8430]</td>
</tr>
</tbody>
</table>
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 $\log L(\beta(k))$ 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 $\hat{\beta}$.

Pol.G()
Note that our implementation of the Newton-Raphson algorithm is rather basic for more robust implementations see, for example, scipy.optimize

4.4.6 Maximum Likelihood Estimation with statsmodels

Now that we know what's going on under the hood, we can apply MLE to an interesting application.

Well use the Poisson regression model in statsmodels to obtain 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, lets re-estimate our simple model with statsmodels to confirm we obtain the same coefficients and log-likelihood value.

```python
from statsmodels.api import Poisson
from scipy import stats
stats.chisqprob = lambda chisq, df: stats.chi2.sf(chisq, df)

X = np.array([[1, 2, 5],
              [1, 3],
              [4, 2]],
```
```python
[[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
==============================================================================
Model:              Poisson   Df Residuals:      2
Method:             MLE        Df Model:         2
Date:               Wed, 26 Jul 2017   Pseudo R-squ.:  0.2546
converged:                  True   LL-Null:    -4.5325
                             LLR p-value:  0.3153
==============================================================================
                          coef     std err      z    P>|z|      [0.025   0.975]
------------------------------------------------------------------------------
const              -6.0785    5.2790    -1.151  0.250     -16.425   4.268
x1                 0.9334    0.8290     1.126  0.260     -0.691    2.558
x2                 0.8433    0.7980     1.057  0.291     -0.720    2.407
==============================================================================
```

Now lets replicate results from Daniel Treismans paper, *Russias Billionaires*, mentioned earlier in the lecture

Treisman starts by estimating equation (4.1), 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)

```python
# Keep only year 2008
df = df[df['year'] == 2008]

# Add a constant
df['const'] = 1

# Variable sets
reg1 = ['const', 'lngdppc', 'lnpop', 'gatttwo08']
```

### 4.4. Maximum Likelihood Estimation 319
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

Well use robust standard errors as in the authors paper

```
import statsmodels.api as sm

# Specify model
poisson_reg = sm.Poisson(df[['numbil0']], df[reg1],
                         missing='drop').fit(cov_type='HC0')
print(poisson_reg.summary())
```

Warning: Maximum number of iterations has been exceeded.

```
Warning: Maximum number of iterations has been exceeded.
Current function value: 2.226090
Iterations: 35
```

Poisson Regression Results
```
==============================================================================
Dep. Variable: numbil0  No. Observations: 197
Model: Poisson  Df Residuals: 193
Method: MLE  Df Model: 3
Date: Wed, 26 Jul 2017  Pseudo R-squ.: 0.8574
Time: 15:41:38  Log-Likelihood: -438.54
converged: False  LL-Null: -3074.7
LLR p-value: 0.000
==============================================================================

coef  std err  z     P>|z|    [0.025   0.975]
const  -29.05  2.58  -11.27  0.000  -34.10  -23.99
lngdppc  1.08  0.14  7.83  0.000   0.81  1.35
lnpop   1.17  0.09  12.02  0.000   0.98  1.36
gattwto08  0.01  0.01  0.98  0.328  -0.01   0.02
==============================================================================
```

Here we received a warning message saying Maximum number of iterations has been exceeded.

Lets try increasing the maximum number of iterations that the algorithm is allowed (the .fit() docstring tells us the default number of iterations is 35)

```
poisson_reg = sm.Poisson(df[['numbil0']], df[reg1],
                         missing='drop').fit(cov_type='HC0', maxiter=100)
print(poisson_reg.summary())
```

Optimization terminated successfully.

```
Optimization terminated successfully.
Current function value: 2.226090
Iterations 36
```

Poisson Regression Results
```
==============================================================================
Dep. Variable: numbil0  No. Observations: 197
Model: Poisson  Df Residuals: 193
Method: MLE  Df Model: 3
```
Success! The algorithm was able to achieve convergence in 36 iterations

Our output indicates that GDP per capita, population, and years of membership in the General Agreement in Tariffs and Trade (GATT) are positively related to the number of billionaires a country has, as expected

Lets also estimate the authors more full-featured models and display them in a single table

```python
from statsmodels.iolib.summary2 import summary_col

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')
```

4.4. Maximum Likelihood Estimation
The output suggests that the frequency of billionaires is positively correlated with GDP per capita, population size, stock market capitalization, and negatively correlated with top marginal income tax rate.

To analyze our results by country, we can plot the difference between the predicted and actual values, then sort from highest to lowest and plot the first 15:

```
print(results_table)

Table 1 - Explaining the Number of Billionaires in 2008
==================================================
<table>
<thead>
<tr>
<th></th>
<th>Model 1</th>
<th>Model 2</th>
<th>Model 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>-29.050***</td>
<td>-19.444***</td>
<td>-20.858***</td>
</tr>
<tr>
<td></td>
<td>(2.578)</td>
<td>(4.820)</td>
<td>(4.255)</td>
</tr>
<tr>
<td>lngdppc</td>
<td>1.084***</td>
<td>0.717***</td>
<td>0.737***</td>
</tr>
<tr>
<td></td>
<td>(0.138)</td>
<td>(0.244)</td>
<td>(0.233)</td>
</tr>
<tr>
<td>lnpop</td>
<td>1.171***</td>
<td>0.806***</td>
<td>0.929***</td>
</tr>
<tr>
<td></td>
<td>(0.097)</td>
<td>(0.213)</td>
<td>(0.195)</td>
</tr>
<tr>
<td>gattwto08</td>
<td>0.006</td>
<td>0.007</td>
<td>0.004</td>
</tr>
<tr>
<td></td>
<td>(0.007)</td>
<td>(0.006)</td>
<td>(0.006)</td>
</tr>
<tr>
<td>lnmcap08</td>
<td>0.399**</td>
<td>0.286*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.172)</td>
<td>(0.167)</td>
<td></td>
</tr>
<tr>
<td>rintr</td>
<td>-0.010</td>
<td>-0.009</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.010)</td>
<td>(0.010)</td>
<td></td>
</tr>
<tr>
<td>topint08</td>
<td>-0.051***</td>
<td>-0.058***</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.011)</td>
<td>(0.012)</td>
<td></td>
</tr>
<tr>
<td>nrrents</td>
<td>-0.005</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.010)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>roflaw</td>
<td>0.203</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.372)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pseudo R-squared</td>
<td>0.86</td>
<td>0.90</td>
<td>0.90</td>
</tr>
<tr>
<td>No. observations</td>
<td>197</td>
<td>131</td>
<td>131</td>
</tr>
</tbody>
</table>
==================================================

Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
```

```python
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=(12,3),
                      legend=False)
plt.ylabel('Number of billionaires above predicted level')```
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.

### 4.4.7 Summary

In this lecture we used Maximum Likelihood Estimation to estimate the parameters of a Poisson model.

`statsmodels` contains other built-in likelihood models such as `Probit` and `Logit`.

For further flexibility, `statsmodels` provides a way to specify the distribution manually using the `GenericLikelihoodModel` class - an example notebook can be found [here](#).
4.4.8 Exercises

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

\[
 f(y_i; \beta) = \mu_i^{y_i}(1 - \mu_i)^{1-y_i}, \quad y_i = 0, 1
\]

where \( \mu_i = \Phi(x_i'\beta) \)

\( \Phi \) represents the *cumulative normal distribution* and constrains the predicted \( y_i \) to be between 0 and 1 (as required for a probability).

\( \beta \) 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.

Exercise 2

Use the following dataset and initial values of \( \beta \) to estimate the MLE with the Newton-Raphson algorithm developed earlier in the lecture:

\[
 X = \begin{bmatrix}
 1 & 2 & 4 \\
 1 & 1 & 1 \\
 1 & 4 & 3 \\
 1 & 5 & 6 \\
 1 & 3 & 5
\end{bmatrix}, \quad y = \begin{bmatrix}
 1 \\
 0 \\
 1 \\
 0 \\
 0
\end{bmatrix}, \quad \beta(0) = \begin{bmatrix}
 1 \\
 0.1 \\
 0.1
\end{bmatrix}
\]

Verify your results with *statsmodels* - you can import the Probit function with the following import statement:

```python
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.

4.4.9 Solutions

Exercise 1

The log-likelihood can be written as

\[
 \log \mathcal{L} = \sum_{i=1}^{n} [y_i \log \Phi(x_i'\beta) + (1 - y_i) \log (1 - \Phi(x_i'\beta))]
\]
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 \( \phi \) is the marginal normal distribution.

The gradient vector of the Probit model is

\[
\frac{\partial \log L}{\partial \beta} = \sum_{i=1}^{n} \left[ y_i \frac{\phi(x_i' \beta)}{\Phi(x_i' \beta)} - (1 - y_i) \frac{1}{1 - \Phi(x_i' \beta)} \right] x_i
\]

The Hessian of the Probit model is

\[
\frac{\partial^2 \log L}{\partial \beta^2} = -\sum_{i=1}^{n} \phi(x_i' \beta) \left[ y_i \frac{\phi(x_i' \beta)}{\Phi(x_i' \beta)^2} + (1 - y_i) \frac{\phi(x_i' \beta) - x_i' \beta (1 - \Phi(x_i' \beta))}{[1 - \Phi(x_i' \beta)^2]} \right] x_i x_i'
\]

Using these results, we can write a class for the Probit model as follows

```python
from scipy.stats import norm

class ProbitRegression:
    def __init__(self, y, X, beta):
        self.X, self.y, self.beta = X, y, beta
        self.n, self.k = X.shape

    def mu(self):
        return norm.cdf(self.X @ self.beta.T)

    def grad(self):
        return norm.pdf(self.X @ self.beta.T)

    def logL(self):
        mu = self.mu()
        return np.sum(y * np.log(mu) + (1 - y) * np.log(1 - mu))

    def H(self):
        X = self.X
        beta = self.beta
        mu = self.mu()
        a = (X @ beta.T) * mu / mu**2
        b = (X @ beta.T) * (1 - mu) / (1 - mu)**2
        return -(y * a + (1 - y) * b) * X.T @ X
```

4.4. Maximum Likelihood Estimation
Exercise 2

\[ X = \text{np.array([[1, 2, 4],} \]
\[ [1, 1, 1],} \]
\[ [1, 4, 3],} \]
\[ [1, 5, 6],} \]
\[ [1, 3, 5]])} \]
\[ y = \text{np.array([1, 0, 1, 1, 0])} \]

# Take a guess at initial \( \beta \)s
\[ \beta = \text{np.array([0.1, 0.1, 0.1])} \]

# Create instance of Probit regression class
prob = \text{ProbitRegression(y, X, \beta)}

# Run Newton-Raphson algorithm
\text{newton_raphson(prob)}

\begin{verbatim}
Iteration_k  Log-likelihood  \( \theta \)
-----------------------------
0  -2.37968841  [-1.3400, 0.7750, -0.1570]
1  -2.36875259  [-1.5350, 0.7750, -0.0980]
2  -2.36872942  [-1.5460, 0.7780, -0.0970]
3  -2.36872942  [-1.5460, 0.7780, -0.0970]
Number of iterations: 4
\( \hat{\beta} \) = [-1.54625858, 0.77778952, -0.09709757]
\end{verbatim}

\[ \text{array([-1.54625858, 0.77778952, -0.09709757])} \]

# Use statsmodels to verify results
\text{print(Probit(y, X).fit().summary())}

Optimization terminated successfully.
Current function value: 0.473746
Iterations 6
Probit Regression Results

\begin{verbatim}
Model:  Probit  Df Residuals:  2
Method:  MLE  Df Model:  2
Date:  Wed, 26 Jul 2017  Pseudo R-squ.:  0.2961
converged:  True  LL-Null:  -3.3651
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      | 0.77778 | 0.788 | 0.986 | 0.324 | -0.768  | 2.323 |
| x2      | -0.0971 | 0.590 | -0.165 | 0.869 | -1.254  | 1.060 |
\end{verbatim}
This section of the course contains foundational mathematical and statistical tools and techniques.

### 5.1 Linear Algebra

#### Contents

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

#### 5.1.1 Overview

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

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

\[
\begin{align*}
y_1 &= ax_1 + bx_2 \\
y_2 &= cx_1 + dx_2
\end{align*}
\]

or, more generally,
\[ y_1 = a_{11}x_1 + a_{12}x_2 + \cdots + a_{1k}x_k \]
\[ \vdots \]
\[ y_n = a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nk}x_k \]

(5.1)

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.

### 5.1.2 Vectors

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

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

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

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

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

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

The following figure represents three vectors in this manner.

```python
import matplotlib.pyplot as plt

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()

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:

\[
\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{pmatrix}
\]
Scalar multiplication is an operation that takes a number $\gamma$ 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

```python
import numpy as np

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:

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

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

4 * x

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

```python
np.sum(x * y)  # Inner product of x and y
12.0
np.sqrt(np.sum(x * x))  # Norm of x, take one
1.7320508075688772
np.linalg.norm(x)  # Norm of x, take two
1.7320508075688772
```

Span

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

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

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

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

for some scalars $\beta_1, \ldots, \beta_k$.

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

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

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

The span is a 2 dimensional plane passing through these two points and the origin.

Chapter 5. Tools and Techniques
```python
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
from scipy.interpolate import interp2d

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()
```

5.1. Linear Algebra
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

\[
x = x_1e_1 + x_2e_2 + x_3e_3
\]

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 \( \mathbb{R}^3 \)
Linear Independence

As well see, its 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 called linear independence.

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

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

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

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

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

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

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

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

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

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

Unique Representations

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

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

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

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

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

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

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

5.1. Linear Algebra 337
5.1.3 Matrices

Matrices are a neat way of organizing data for use in linear operations. An $n \times k$ matrix is a rectangular array $A$ of numbers with $n$ rows and $k$ columns:

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

Often, the numbers in the matrix represent coefficients in a system of linear equations, as discussed at the start of this lecture. For obvious reasons, the matrix $A$ is also called a vector if either $n = 1$ or $k = 1$. In the former case, $A$ is called a row vector, while in the latter it is called a column vector.

If $n = k$, then $A$ is called square. The matrix formed by replacing $a_{ij}$ by $a_{ji}$ for every $i$ and $j$ is called the transpose of $A$, and denoted $A'$ or $A^\top$.

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

For a square matrix $A$, the $i$ elements of the form $a_{ii}$ for $i = 1, \ldots, n$ are called the principal diagonal. $A$ is called diagonal if the only nonzero entries are on the principal diagonal. If, in addition to being diagonal, each element along the principal diagonal is equal to 1, then $A$ is called the identity matrix, and denoted by $I$.

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:

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

and

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

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

\[
Ax = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} := \begin{bmatrix} a_{11}x_1 + \cdots + a_{1k}x_k \\ \vdots \\ a_{n1}x_1 + \cdots + a_{nk}x_k \end{bmatrix}
\]  

(5.2)

**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 \( IA = A \).

**Matrices in NumPy**

NumPy arrays are also used as matrices, and have fast, efficient functions and methods for all the standard matrix operations\(^1\).

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

\[
A = ((1, 2), \\
(3, 4))
\]

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

type(A)
```

tuple

```python
A = np.array(A)

type(A)
```

```python
numpy.ndarray
```

\(^1\) Although there is a specialized matrix data type defined in NumPy, its more standard to work with ordinary NumPy arrays. See this discussion.
A.shape

(2, 2)

The `shape` attribute is a tuple giving the number of rows and columns. See [here](#) for more discussion.

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

There are many convenient functions for creating common matrices (matrices of zeros, ones, etc.) See [here](#).

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

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

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

```python
A + B
```

```python
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.

### 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 $f: \mathbb{R}^k \to \mathbb{R}^n$ is called **linear** if, for all $x, y \in \mathbb{R}^k$ and all scalars $\alpha, \beta$, 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 $x$.

### 5.1.4 Solving Systems of Equations

Recall again the system of equations (5.1).
If we compare (5.1) and (5.2), we see that (5.1) can now be written more conveniently as

\[ y = Ax \]  

(5.3)

The problem we face is to determine a vector \( x \in \mathbb{R}^k \) that solves (5.3), taking \( y \) and \( A \) as given.

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

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

If so, is it always unique?

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

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

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

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

5.1. Linear Algebra
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 (5.3) that rule out these problems?

In this context, the most important thing to recognize about the expression $Ax$ is that it corresponds to a linear combination of the columns of $A$
In particular, if \( a_1, \ldots, a_k \) are the columns of \( A \), then
\[
Ax = x_1a_1 + \cdots + x_ka_k
\]
Hence the range of \( f(x) = Ax \) is exactly the span of the columns of \( A \).

We want the range to be large, so that it contains arbitrary \( y \).

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

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

Indeed, it follows from our *earlier discussion* that if \( \{a_1, \ldots, a_k\} \) are linearly independent and \( y = Ax = x_1a_1 + \cdots + x_ka_k \), then no \( z \neq x \) satisfies \( y = Az \).

**The \( n \times n \) 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 were 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.

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, 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 a 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.

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 $\mathbb{R}^2$. 

344 Chapter 5. Tools and Techniques
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
\[
a_1 = a_2 + a_3
\]
Then if \( y = Ax = x_1a_1 + x_2a_2 + x_3a_3 \), we can also write
\[
y = x_1(\alpha a_2 + \beta a_3) + x_2a_2 + x_3a_3 = (x_1\alpha + x_2)a_2 + (x_1\beta + x_3)a_3
\]
In other words, uniqueness fails

**Linear Equations with SciPy**

Here's an illustration of how to solve linear equations with SciPy's \texttt{linalg} submodule
All of these routines are Python front ends to time-tested and highly optimized FORTRAN code

```python
from scipy.linalg import inv, solve, det
A = ((1, 2), (3, 4))
A = np.array(A)
y = np.ones((2, 1))  # Column vector
det(A)  # Check that A is nonsingular, and hence invertible

-2.0

A_inv = inv(A)  # Compute the inverse
A_inv

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

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

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

solve(A, y)  # Produces same solution

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

Observe how we can solve for \( x = A^{-1}y \) by either via \texttt{inv(A) \@ y}, or using \texttt{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 \texttt{scipy.linalg.lstsq(A, y)}
5.1.5 Eigenvalues and Eigenvectors

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

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

$$Av = \lambda v$$

then we say that $\lambda$ 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.

```python
from scipy.linalg import eig

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)
```
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 \(\lambda\) 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 \(\lambda\) of degree \(n\).

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

Some nice facts about the eigenvalues of a square matrix \(A\) are as follows:

1. The determinant of \(A\) equals the product of the eigenvalues.

2. The trace of \(A\) (the sum of the elements on the principal diagonal) equals the sum of the eigenvalues.
3. If $A$ is symmetric, then all of its eigenvalues are real

4. If $A$ is invertible and $\lambda_1, \ldots, \lambda_n$ are its eigenvalues, then the eigenvalues of $A^{-1}$ are $1/\lambda_1, \ldots, 1/\lambda_n$

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

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

```python
A = ((1, 2),
     (2, 1))
A = np.array(A)
evals, evecs = eig(A)
evals

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

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

Note that the columns of `evecs` are the eigenvectors

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

**Generalized Eigenvalues**

It is sometimes useful to consider the *generalized eigenvalue problem*, which, for given matrices $A$ and $B$, seeks generalized eigenvalues $\lambda$ 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

### 5.1.6 Further Topics

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

**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.

Neumanns 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 $A$.

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

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k$$  (5.4)

Spectral Radius

A result known as Gelfands 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 Gelfands formula, if all eigenvalues are strictly less than one in modulus, there exists a $k$ with $\|A^k\| < 1$.

In which case (5.4) is valid.

Positive Definite Matrices

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

We say that $A$ is

1. positive definite if $x'Ax > 0$ for every $x \in \mathbb{R}^n \setminus \{0\}$

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

$^2$ 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.
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).

### Differentiating Linear and Quadratic forms

The following formulas are useful in many economic contexts. Let

- $z, x$ and $a$ all be $\mathbb{R}^n$ 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. $\frac{\partial y'Bz}{\partial B} = y'z$

Exercise 1 below asks you to apply these formulas.

### 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 dont mind a slightly abstract approach, a nice intermediate-level text on linear algebra is *Janich94*.

### 5.1.7 Exercises

**Exercise 1**

Let $x$ be a given $\mathbb{R}^n$ vector and consider the problem

$$v(x) = \max_{y,u} \left\{ -y'P - u'Q - y'P - u'Q \right\}$$

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

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

where $\lambda$ 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 $L$ with respect to $y,u$ and minimizing it with respect to $\lambda$

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 dont care about the Lagrange multipliers, we can substitute the constraint into the objective function, and then just maximize $-(Ax + Bu)'P(Ax + Bu) - u'Q u$ with respect to $u$. You can verify that this leads to the same maximizer.

### 5.1.8 Solutions

**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.

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 \( u = -(Q + BPB)^{-1}BPAX \), then

\[v(x) = -(Ax + Bu)'P(Ax + Bu) - u'Qu \quad \text{with} \quad u = -(Q + B'PB)^{-1}B'PAx\]
To evaluate the function

\[ v(x) = -(Ax + Bu)'P(Ax + Bu) - u'Qu \]
\[ = -(x'A' + u'B')P(Ax + Bu) - u'Qu \]
\[ = -x'A'PAx - u'B'PAx - x'A'PBu - u'B'PBu - u'Qu \]
\[ = -x'A'PAx - 2u'B'PAx - u'(Q + B'PB)u \]

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

Regarding the second term \(-2u'B'PAx\),

\[ -2u'B'PAx = -2x'S'B'PAx \]
\[ = 2x'A'PB(Q + B'PB)^{-1}B'PAx \]

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

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

\[ -u'(Q + B'PB)u = -x'S'(Q + B'PB)Sx \]
\[ = -x'A'PB(Q + B'PB)^{-1}B'PAx \]

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

This implies that

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

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

5.2 Orthogonal Projections and Their Applications
### 5.2.1 Overview

Orthogonal projection is a cornerstone of vector space methods, with many diverse applications. These include, but are not limited to,

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

In this lecture we focus on

- key ideas
- least squares regression

### Further Reading

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

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

For a complete set of proofs in a general setting, see, for example, [Rom05]

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

### 5.2.2 Key Definitions

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

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

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

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

When $\langle x, z \rangle = 0$, then $\cos(\theta) = 0$ and $x$ and $z$ are said to be orthogonal and we write $x \perp z$
For a linear subspace $S \subseteq \mathbb{R}^n$, we call $x \in \mathbb{R}^n$ \textbf{orthogonal to} $S$ if $x \perp z$ for all $z \in S$, and write $x \perp S$.
The **orthogonal complement** of linear subspace $S \subseteq \mathbb{R}^n$ 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 $\{x_1, \ldots, x_k\} \subset \mathbb{R}^n$ is called an **orthogonal set** if $x_i \perp x_j$ whenever $i \neq j$

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

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

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

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

### 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 well *see below*

---

5.2. Orthogonal Projections and Their Applications
5.2.3 The Orthogonal Projection Theorem

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

The next theorem provides answers this question

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

\[
\hat{y} := \arg \min_{z \in S} \| y - z \|
\]

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

- \( \hat{y} \in S \)
- \( y - \hat{y} \perp S \)

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

The next figure provides some intuition

**Proof of sufficiency**

Well 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

**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 \( \bar{E}_S y = Py \), where \( \bar{E} \) 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 \( y \in \mathbb{R}^n \),

\[ 5.2. \text{Orthogonal Projections and Their Applications} \]
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

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

The next figure illustrates
5.2.4 Orthogonal Basis

An orthogonal set of vectors $O \subset \mathbb{R}^n$ is called an \textbf{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 $\text{span} O = S$, then $O$ is called an \textbf{orthonormal basis} of $S$

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

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

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

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

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

$$x = \sum_{j=1}^{k} \alpha_j u_j \quad (5.5)$$
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 (5.5) verifies the claim.

**Projection onto an Orthonormal Basis**

When the subspace onto which we are projecting is orthonormal, computing the projection simplifies:

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

$$Py = \sum_{i=1}^{k} \langle y, u_i \rangle u_i, \quad \forall y \in \mathbb{R}^n \quad (5.6)$$

Proof: Fix $y \in \mathbb{R}^n$ and let $Py$ be defined as in (5.6)

Clearly, $Py \in S$

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

It suffices to show that $y - Py \perp$ any basis vector $u_i$ (why?)

This is true because

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

**5.2.5 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](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.

**Starting with \( X \)**

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

\[ S := \text{span} \, X := \text{span}\{\col_1 X, \ldots, \col_k X\} \]

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**

**The Orthonormal Case**

Suppose that \( U \) is \( n \times k \) with orthonormal columns

Let \( u_i := \col U_i \) for each \( i \), let \( S := \text{span} \, U \) 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.
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 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

**5.2.6 Least Squares Regression**

Let 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

**Squared risk measures**

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

\[
R(f) := \mathbb{E} [(y - f(x))^2]
\]
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**.

**Solution**

Define the matrices

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

and

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

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

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

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

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

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

$$P := X(X'X)^{-1}X' \quad \text{and} \quad M := I - P$$
The vector of fitted values is
\[ \hat{y} := X \hat{\beta} = Py \]

The vector of residuals is
\[ \hat{u} := y - \hat{y} = y - Py = My \]

Here are some more standard definitions:

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

\[ \text{TSS} = \text{ESS} + \text{SSR} \]

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

### 5.2.7 Orthogonalization and Decomposition

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

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

The next section gives details

**Gram-Schmidt Orthogonalization**

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

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

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

One description of this procedure is as follows:

- For \( i = 1, \ldots, k \), form \( S_i := \text{span}\{x_1, \ldots, x_i\} \) and \( S_i^\perp \)

- Set \( v_1 = x_1 \)

- For \( i \geq 2 \) set \( v_i := E_{S_{i-1}^\perp} x_i \) and \( u_i := v_i / \|v_i\| \)

The sequence \( u_1, \ldots, u_k \) has the stated properties

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

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

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

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

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

Some rearranging gives \( X = QR \)

### 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

\[
\hat{\beta} = (R'Q'R)^{-1}R'Q'y = (R'R)^{-1}R'Q'y = R^{-1}(R')^{-1}R'Q'y = R^{-1}Q'y
\]

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

### 5.2.8 Exercises

**Exercise 1**

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

**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?
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

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

5.2.9 Solutions

Exercise 1

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

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.)

Exercise 3

Here is a function that computes the orthonormal vectors using the GS algorithm given in the lecture.

```python
import numpy as np

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
    """
```
Here are the arrays well work with

```python
y = [1, 3, -3]
X = [[1, 0], [0, -6], [2, 2]]
X, y = [np.asarray(z) for z in (X, y)]
```

First lets try projection of $y$ onto the column space of $X$ using the ordinary matrix expression:

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

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

Now lets do the same using an orthonormal basis created from our `gram_schmidt` function.

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

```
array([[ 0.4472136 , -0.13187609], [ 0. , -0.98907071], [ 0.89442719, 0.06593805]])
```
This is the same answer. So far so good. Finally, let's try the same thing but with the basis obtained via QR decomposition:

```python
from scipy.linalg import qr
Q, R = qr(X, mode='economic')
Q
```

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

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

```text
array([-0.56521739,  3.26086957, -2.21739131])
```

Again, we obtain the same answer.

## 5.3 LLN and CLT

### Contents

- **LLN and CLT**
  - Overview
  - Relationships
  - LLN
  - CLT
  - Exercises
  - Solutions

### 5.3.1 Overview

This lecture illustrates two of the most important theorems of probability and statistics: The law of large numbers (LLN) and the central limit theorem (CLT).
These beautiful theorems lie behind many of the most fundamental results in econometrics and quantitative economic modeling.

The lecture is based around simulations that show the LLN and CLT in action.

We also demonstrate how the LLN and CLT break down when the assumptions they are based on do not hold.

In addition, we examine several useful extensions of the classical theorems, such as:

- The delta method, for smooth functions of random variables
- The multivariate case

Some of these extensions are presented as exercises.

### 5.3.2 Relationships

The CLT refines the LLN.

The LLN gives conditions under which sample moments converge to population moments as sample size increases.

The CLT provides information about the rate at which sample moments converge to population moments as sample size increases.

### 5.3.3 LLN

We begin with the law of large numbers, which tells us when sample averages will converge to their population means.

**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 *Kolmogorovs strong law*.

Let $X_1, \ldots, X_n$ be independent and identically distributed scalar random variables, with common distribution $F$.

When it exists, let $\mu$ denote the common mean of this sample:

$$
\mu := \mathbb{E}X = \int xf(x) \, dx
$$

In addition, let

$$
\bar{X}_n := \frac{1}{n} \sum_{i=1}^{n} X_i
$$

Kolmogorovs strong law states that, if $\mathbb{E}|X|$ is finite, then...
\[ P \{ \bar{X}_n \to \mu \text{ as } n \to \infty \} = 1 \tag{5.7} \]

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, (5.7) 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.

**Proof**

The proof of Kolmogorov’s strong law is nontrivial – see, for example, theorem 8.3.5 of [Dud02].

On the other hand, we can prove a weaker version of the LLN very easily and still get most of the intuition.

The version we prove is as follows: If \( X_1, \ldots, X_n \) is IID with \( E X_i^2 < \infty \), then, for any \( \epsilon > 0 \), we have

\[ P \{ |\bar{X}_n - \mu| \geq \epsilon \} \to 0 \text{ as } n \to \infty \tag{5.8} \]

(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 \( \sigma^2 \) be the variance of each \( X_i \).

Recall the Chebyshev inequality, which tells us that

\[ P \{ |\bar{X}_n - \mu| \geq \epsilon \} \leq \frac{\mathbb{E}[(\bar{X}_n - \mu)^2]}{\epsilon^2} \tag{5.9} \]

Now observe that

\[
\mathbb{E}[(\bar{X}_n - \mu)^2] = \mathbb{E} \left\{ \left( \frac{1}{n} \sum_{i=1}^{n} (X_i - \mu) \right)^2 \right\} \\
= \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \mathbb{E}(X_i - \mu)(X_j - \mu) \\
= \frac{1}{n^2} \sum_{i=1}^{n} \mathbb{E}(X_i - \mu)^2 \\
= \frac{\sigma^2}{n}
\]

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 (5.9), we come to the estimate

\[
P\{ |\bar{X}_n - \mu| \geq \epsilon \} \leq \frac{\sigma^2}{n\epsilon^2} \tag{5.10}
\]

The claim in (5.8) is now clear

Of course, if the sequence \( X_1, \ldots, X_n \) is correlated, then the cross-product terms \( \mathbb{E}(X_i - \mu)(X_j - \mu) \) are not necessarily zero

While this doesn't mean that the same line of argument is impossible, it does mean that if we want a similar result then the covariances should be almost zero for most of these terms

In a long sequence, this would be true if, for example, \( \mathbb{E}(X_i - \mu)(X_j - \mu) \) approached zero when the difference between \( i \) and \( j \) became large

In other words, the LLN can still work if the sequence \( X_1, \ldots, X_n \) has a kind of asymptotic independence, in the sense that correlation falls to zero as variables become further apart in the sequence

This idea is very important in time series analysis, and well come across it again soon enough

**Illustration**

Lets now illustrate the classical IID law of large numbers using simulation

In particular, we aim to generate some sequences of IID random variables and plot the evolution of \( \bar{X}_n \) as \( n \) increases

Below is a figure that does just this (as usual, you can click on it to expand it)

It shows IID observations from three different distributions and plots \( \bar{X}_n \) against \( n \) in each case

The dots represent the underlying observations \( X_i \) for \( i = 1, \ldots, 100 \)

In each of the three cases, convergence of \( \bar{X}_n \) to \( \mu \) occurs as predicted

```python
import random
import numpy as np
from scipy.stats import t, beta, lognorm, expon, gamma, poisson
import matplotlib.pyplot as plt

n = 100

# == Arbitrary collection of distributions == #
distributions = {
    "student's t with 10 degrees of freedom": t(10),
    "\beta(2, 2)": beta(2, 2),
    "lognormal LN(0, 1/2)": lognorm(0.5),
    "\gamma(5, 1/2)": gamma(5, scale=2),
    "poisson(4)": poisson(4),
    "exponential with \lambda = 1": expon(1)}

# == Create a figure and some axes == #
```
```python
num_plots = 3
dist = plt.subplots(num_plots, 1, figsize=(20, 20))

bbox = (0., 1.02, 1., .102)
legend_args = {'ncol': 2,
               'bbox_to_anchor': bbox,
               'loc': 3,
               'mode': 'expand'}
dist.subplots_adjust(hspace=0.5)

for ax in dist[1]:
    name = random.choice(list(distributions.keys()))
distribution = distributions.pop(name)

data = distribution.rvs(n)

sample_mean = np.empty(n)
for i in range(n):
    sample_mean[i] = np.mean(data[:i+1])

ax.plot(list(range(n)), data, 'o', color='grey', alpha=0.5)
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)

plt.show()
```
The three distributions are chosen at random from a selection stored in the dictionary `distributions`

### Infinite Mean

What happens if the condition $E|X| < \infty$ in the statement of the LLN is not satisfied?

This might be the case if the underlying distribution is heavy tailed. The best known example is the Cauchy distribution, which has density

$$f(x) = \frac{1}{\pi(1 + x^2)} \quad (x \in \mathbb{R})$$

The next figure shows 100 independent draws from this distribution.
from scipy.stats import cauchy

n = 100
distribution = cauchy()

fig, ax = plt.subplots(figsize=(10, 6))
data = distribution.rvs(n)

ax.plot(list(range(n)), data, linestyle='-', marker='o', alpha=0.5)
ax.vlines(list(range(n)), 0, data, lw=0.2)
ax.set_title(f"{n} observations from the Cauchy distribution")
plt.show()

Notice how extreme observations are far more prevalent here than the previous figure

Let's now have a look at the behavior of the sample mean

n = 1000
distribution = cauchy()

fig, ax = plt.subplots(figsize=(10, 6))
data = distribution.rvs(n)

# == Compute sample mean at each n ==#
sample_mean = np.empty(n)
Here we've increased $n$ to 1000, but the sequence still shows no sign of converging

Will convergence become visible if we take $n$ even larger?

The answer is no

To see this, recall that the characteristic function of the Cauchy distribution is

$$
\phi(t) = \mathbb{E}e^{itX} = \int e^{itx} f(x) dx = e^{-|t|}
$$

(5.11)
Using independence, the characteristic function of the sample mean becomes

\[
E e^{itX_n} = \mathbb{E} \exp \left\{ \frac{t}{n} \sum_{j=1}^{n} X_j \right\} = \mathbb{E} \prod_{j=1}^{n} \exp \left\{ i \frac{t}{n} X_j \right\} = \prod_{j=1}^{n} \mathbb{E} \exp \left\{ i \frac{t}{n} X_j \right\} = [\phi(t/n)]^n
\]

In view of (5.11), 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 a point

### 5.3.4 CLT

Next we turn to the central limit theorem, which tells us about the distribution of the deviation between sample averages and population means

#### Statement of the Theorem

The central limit theorem is one of the most remarkable results in all of mathematics

In the classical IID setting, it tells us the following:

If the sequence \( X_1, \ldots, X_n \) is IID, with common mean \( \mu \) and common variance \( \sigma^2 \in (0, \infty) \), then

\[
\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} N(0, \sigma^2) \quad \text{as} \quad n \to \infty
\]

(5.12)

Here \( \xrightarrow{d} N(0, \sigma^2) \) indicates convergence in distribution to a centered (i.e., zero mean) normal with standard deviation \( \sigma \)

#### Intuition

The striking implication of the CLT is that for any distribution with finite second moment, the simple operation of adding independent copies always leads to a Gaussian curve

A relatively simple proof of the central limit theorem can be obtained by working with characteristic functions (see, e.g., theorem 9.5.6 of [Dud02])

The proof is elegant but almost anticlimactic, and it provides surprisingly little intuition

In fact all of the proofs of the CLT that we know are similar in this respect
Why does adding independent copies produce a bell-shaped distribution?

Part of the answer can be obtained by investigating addition of independent Bernoulli random variables. In particular, let $X_i$ be binary, with $P(X_i = 0) = P(X_i = 1) = 0.5$, and let $X_1, \ldots, X_n$ be independent. Think of $X_i = 1$ as a success, so that $Y_n = \sum_{i=1}^{n} X_i$ is the number of successes in $n$ trials.

The next figure plots the probability mass function of $Y_n$ for $n = 1, 2, 4, 8$.

```python
from scipy.stats import binom

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 ever more pronounced

We are witnessing the binomial approximation of the normal distribution

### 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 finite second moment)

```python
from scipy.stats import norm

# == Set parameters == #
n = 250                  # Choice of n
k = 1000000              # Number of draws of Y_n
distribution = expon(2)  # Exponential distribution, \( \lambda = 1/2 \)
\mu, 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 ==#
```

---

380 Chapter 5. Tools and Techniques
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, normed=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 program produces figures such as the one below

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 $F$

5.3. LLN and CLT
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 well 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 $Y_5$

```python
from scipy.stats import gaussian_kde
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.collections import PolyCollection

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)
```
```python
# == 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")
ax.invert_xaxis()
ax.view_init(30, 45)  # Rotates the plot 30 deg on z axis and 45 deg on x axis
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.

**The Multivariate Case**

The law of large numbers and central limit theorem work just as nicely in multidimensional settings.

To state the results, let’s recall some elementary facts about random vectors.

A random vector \( \mathbf{X} \) is just a sequence of \( k \) random variables \((X_1, \ldots , X_k)\)

Each realization of \( \mathbf{X} \) is an element of \( \mathbb{R}^k \)

A collection of random vectors \( \mathbf{X}_1, \ldots , \mathbf{X}_n \) is called independent if, given any \( n \) vectors \( \mathbf{x}_1, \ldots , \mathbf{x}_n \) in \( \mathbb{R}^k \), we have

\[
P\{ \mathbf{X}_1 \leq \mathbf{x}_1, \ldots , \mathbf{X}_n \leq \mathbf{x}_n \} = P\{ \mathbf{X}_1 \leq \mathbf{x}_1 \} \times \cdots \times P\{ \mathbf{X}_n \leq \mathbf{x}_n \}
\]

(The vector inequality \( \mathbf{X} \leq \mathbf{x} \) means that \( X_j \leq x_j \) for \( j = 1, \ldots , k \))

Let \( \mu_j := \mathbb{E}[X_j] \) for all \( j = 1, \ldots , k \)
The expectation $E[X]$ of $X$ is defined to be the vector of expectations:

$$E[X] := \begin{pmatrix} E[X_1] \\ E[X_2] \\ \vdots \\ E[X_k] \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_k \end{pmatrix} =: \mu$$

The variance-covariance matrix of random vector $X$ is defined as

$$\text{Var}[X] := E[(X - \mu)(X - \mu)']$$

Expanding this out, we get

$$\text{Var}[X] = \begin{pmatrix} E[(X_1 - \mu_1)(X_1 - \mu_1)] & \cdots & E[(X_1 - \mu_1)(X_k - \mu_k)] \\ E[(X_2 - \mu_2)(X_1 - \mu_1)] & \cdots & E[(X_2 - \mu_2)(X_k - \mu_k)] \\ \vdots & \cdots & \vdots \\ E[(X_k - \mu_k)(X_1 - \mu_1)] & \cdots & E[(X_k - \mu_k)(X_k - \mu_k)] \end{pmatrix}$$

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 $X_1, \ldots, X_n$ be a sequence of independent and identically distributed random vectors, each one taking values in $\mathbb{R}^k$

Let $\mu$ be the vector $E[X_i]$, and let $\Sigma$ be the variance-covariance matrix of $X_i$

Interpreting vector addition and scalar multiplication in the usual way (i.e., pointwise), let

$$\overline{X}_n := \frac{1}{n} \sum_{i=1}^{n} X_i$$

In this setting, the LLN tells us that

$$\mathbb{P} \{ \overline{X}_n \to \mu \text{ as } n \to \infty \} = 1 \quad (5.13)$$

Here $\overline{X}_n \to \mu$ means that $\|\overline{X}_n - \mu\| \to 0$, where $\| \cdot \|$ is the standard Euclidean norm

The CLT tells us that, provided $\Sigma$ is finite,

$$\sqrt{n}(\overline{X}_n - \mu) \xrightarrow{d} N(0, \Sigma) \quad \text{as } n \to \infty \quad (5.14)$$

### 5.3.5 Exercises

**Exercise 1**

One very useful consequence of the central limit theorem is as follows
Assume the conditions of the CLT as *stated above*

If \( g : \mathbb{R} \to \mathbb{R} \) is differentiable at \( \mu \) and \( g'(\mu) \neq 0 \), then

\[
\sqrt{n}\{g(\bar{X}_n) - g(\mu)\} \xrightarrow{d} N(0, g'(\mu)^2 \sigma^2) \quad \text{as} \quad n \to \infty \tag{5.15}
\]

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 \( \mu \)

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?

**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. \( X_1, \ldots, X_n \) is a sequence of IID random vectors, each taking values in \( \mathbb{R}^k \)
2. \( \mu := \mathbb{E}[X_i] \), and \( \Sigma \) is the variance-covariance matrix of \( X_i \)
3. The convergence

\[
\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} N(0, \Sigma) \tag{5.16}
\]

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 \( X \) is a random vector in \( \mathbb{R}^k \) and \( A \) is constant and \( k \times k \), then

\[
\text{Var}[AX] = A \text{Var}[X]A'
\]

Second, by the continuous mapping theorem, if \( Z_n \xrightarrow{d} Z \) in \( \mathbb{R}^k \) and \( A \) is constant and \( k \times k \), then

\[
AZ_n \xrightarrow{d} A Z
\]
Third, if $S$ is a $k \times k$ symmetric positive definite matrix, then there exists a symmetric positive definite matrix $Q$, called the inverse square root of $S$, such that

$$QSQ' = I$$

Here $I$ is the $k \times k$ identity matrix.

Putting these things together, your first exercise is to show that if $Q$ is the inverse square root of $\Sigma$, then

$$Z_n := \sqrt{n}Q(\bar{X}_n - \mu) \overset{d}{\to} Z \sim N(0, I)$$

Applying the continuous mapping theorem one more time tells us that

$$\|Z_n\|^2 \overset{d}{\to} \|Z\|^2$$

Given the distribution of $Z$, we conclude that

$$n\|Q(\bar{X}_n - \mu)\|^2 \overset{d}{\to} \chi^2(k)$$

(5.17)

where $\chi^2(k)$ is the chi-squared distribution with $k$ degrees of freedom.

(Recall that $k$ is the dimension of $X_i$, the underlying random vectors)

Your second exercise is to illustrate the convergence in (5.17) with a simulation.

In doing so, let

$$X_i := \left( \begin{array}{c} W_i \\ U_i + W_i \end{array} \right)$$

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 $\Sigma$ from the preceding information.

### 5.3.6 Solutions

**Exercise 1**

Here is one solution.
Illustrates the delta method, a consequence of the central limit theorem.

```python
from scipy.stats import uniform

# == 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, normed=True)
xgrid = np.linspace(xmin, xmax, 200)

lb = "N(0, g'(μ)^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 $\mu$ of this distribution is $\pi/2$, and since $g' = \cos$, we have $g'(\mu) = 0$

Hence the conditions of the delta theorem are not satisfied

**Exercise 2**

First we want to verify the claim that

$$\sqrt{n}Q(X_n - \mu) \xrightarrow{d} N(0, I)$$

This is straightforward given the facts presented in the exercise

Let

$$Y_n := \sqrt{n}(X_n - \mu) \quad \text{and} \quad Y \sim N(0, \Sigma)$$

By the multivariate CLT and the continuous mapping theorem, we have

$$QY_n \xrightarrow{d} QY$$

Since linear combinations of normal random variables are normal, the vector $QY$ is also normal

Its mean is clearly 0, and its variance covariance matrix is

$$\text{Var}[QY] = Q\text{Var}[Y]Q' = Q\Sigma Q' = I$$
In conclusion, $Q Y_n \overset{d}{\to} Q Y \sim N(0, I)$, which is what we aimed to show.

Now we turn to the simulation exercise.

Our solution is as follows:

```python
from scipy.stats import chi2
from scipy.linalg import inv, sqrtm

# == 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 $\Sigma^{-1/2}$ ==#
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)
xb = "Chi-squared with 2 degrees of freedom"
ax.plot(xgrid, chi2.pdf(xgrid, 2), 'k-', lw=2, label=xb)
ax.legend()
ax.hist(chisq_obs, bins=50, normed=True)
plt.show()
```
5.4 Linear State Space Models

We may regard the present state of the universe as the effect of its past and the cause of its future
– Marquis de Laplace
5.4.1 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
  - Friedmans permanent income model of consumption smoothing
  - Barros model of smoothing total tax collections
  - Rational expectations version of Cagans model of hyperinflation
  - Sargent and Wallaces unpleasant monetarist arithmetic, etc.

5.4.2 The Linear State Space Model

The objects in play are:

- An \( n \times 1 \) vector \( x_t \) denoting the state at time \( t = 0, 1, 2, \ldots \)
- An iid sequence of \( m \times 1 \) random vectors \( w_t \sim N(0, I) \)
- A \( k \times 1 \) vector \( y_t \) of observations at time \( t = 0, 1, 2, \ldots \)
- An \( n \times n \) matrix \( A \) called the transition matrix
- An \( n \times m \) matrix \( C \) called the volatility matrix
- A \( k \times n \) matrix \( G \) sometimes called the output matrix

Here is the linear state-space system

\[
\begin{align*}
  x_{t+1} &= Ax_t + Cw_{t+1} \\
  y_t &= Gx_t \\
  x_0 &\sim N(\mu_0, \Sigma_0)
\end{align*}
\]
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 (5.18) 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 well see how to compute these distributions and their moments

Martingale difference shocks

Weve 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

\[
E[w_{t+1}|x_t, x_{t-1}, \ldots] = 0
\]

This is a weaker condition than that \( \{w_t\} \) is iid with \( w_{t+1} \sim N(0, I) \)

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

\[
y_{t+1} = \phi_0 + \phi_1 y_t + \phi_2 y_{t-1} \quad \text{s.t.} \quad y_0, y_{-1} \text{ given} \quad (5.18)
\]

To map (5.18) into our state space system (5.18), we set

\[
x_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix} \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \phi_0 & \phi_1 & \phi_2 \\ 0 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad G = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}
\]
You can confirm that under these definitions, (5.18) and (5.18) agree.

The next figure shows 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 (5.18) to represent the model

$$y_{t+1} = \phi_1 y_t + \phi_2 y_{t-1} + \phi_3 y_{t-2} + \phi_4 y_{t-3} + \sigma w_{t+1}$$  \hspace{1cm} (5.19)

where $\{w_t\}$ is iid and standard normal.

To put this in the linear state space format we take $x_t = [y_t \ y_{t-1} \ y_{t-2} \ y_{t-3}]'$ and

$$A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad G = [1 \ 0 \ 0 \ 0]$$

The matrix $A$ has the form of the *companion matrix* to the vector $[\phi_1 \ \phi_2 \ \phi_3 \ \phi_4]$.

The next figure shows dynamics of this process when $\phi_1 = 0.5, \phi_2 = -0.2, \phi_3 = 0, \phi_4 = 0.5, \sigma = 0.2, y_0 = y_{-1} = y_{-2} = y_{-3} = 1$. 

394 Chapter 5. Tools and Techniques
Vector Autoregressions

Now suppose that

- $y_t$ is a $k \times 1$ vector
- $\phi_j$ is a $k \times k$ matrix and
- $w_t$ is $k \times 1$

Then (5.19) is termed a vector autoregression.

To map this into (5.18), we set

$$x_t = \begin{bmatrix} y_t \\ y_{t-1} \\ y_{t-2} \\ y_{t-3} \end{bmatrix}, \quad A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \end{bmatrix}, \quad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad G = \begin{bmatrix} I & 0 & 0 \end{bmatrix}$$

where $I$ is the $k \times k$ identity matrix and $\sigma$ is a $k \times k$ matrix.

Seasonals

We can use (5.18) 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 (5.19)

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:

\[
x_{t+4} = x_t
\]

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

\[
A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad G = \begin{bmatrix} a \\ b \end{bmatrix}
\]

and starting at initial condition \( x_0 = [0 \quad 1]' \).

In fact it is possible to use the state-space system to represent polynomial trends of any order.

For instance, let

\[
x_0 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix}
\]

It follows that

\[
A^t = \begin{bmatrix} 1 & t & t(t-1)/2 \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix}
\]

Then \( x_t' = [t(t-1)/2 \quad t \quad 1] \), so that \( x_t \) contains linear and quadratic time trends

\(^1\) The eigenvalues of \( A \) are \( (1, -1, i, -i) \).
Moving Average Representations

A nonrecursive expression for $x_t$ as a function of $x_0, w_1, w_2, \ldots, w_t$ can be found by using (5.18) repeatedly to obtain

$$
x_t = Ax_{t-1} + Cw_t
= A^2x_{t-2} + ACw_{t-1} + Cw_t
\vdots
= \sum_{j=0}^{t-1} A^jCw_{t-j} + A^tx_0
$$

Representation (5.21) 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

$$
A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad C = \begin{bmatrix} 1 \\ 0 \end{bmatrix}
$$

You will be able to show that $A^t = \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix}$ and $A^tC = \begin{bmatrix} 1 & 0 \end{bmatrix}'$

Substituting into the moving average representation (5.21), 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**

5.4.3 Distributions and Moments

Unconditional Moments

Using (5.18), its easy to obtain expressions for the (unconditional) means of $x_t$ and $y_t$

Well explain what *unconditional* and *conditional* mean soon

Letting $\mu_t := \mathbb{E}[x_t]$ and using linearity of expectations, we find that
\begin{equation}
\mu_{t+1} = A \mu_t \quad \text{with } \mu_0 \text{ given} \tag{5.21}
\end{equation}

Here \( \mu_0 \) is a primitive given in (5.18)

The variance-covariance matrix of \( x_t \) is \( \Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)'] \)

Using \( x_{t+1} - \mu_{t+1} = A(x_t - \mu_t) + C\epsilon_{t+1} \), we can determine this matrix recursively via

\begin{equation}
\Sigma_{t+1} = A\Sigma_tA' + CC' \quad \text{with } \Sigma_0 \text{ given} \tag{5.22}
\end{equation}

As with \( \mu_0 \), the matrix \( \Sigma_0 \) is a primitive given in (5.18)

As a matter of terminology, we will sometimes call

- \( \mu_t \) the \emph{unconditional mean} of \( x_t \)
- \( \Sigma_t \) the \emph{unconditional variance-covariance matrix} of \( x_t \)

This is to distinguish \( \mu_t \) and \( \Sigma_t \) 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

\begin{equation}
\mathbb{E}[y_t] = \mathbb{E}[Gx_t] = G\mu_t \tag{5.23}
\end{equation}

The variance-covariance matrix of \( y_t \) is easily shown to be

\begin{equation}
\operatorname{Var}[y_t] = \operatorname{Var}[Gx_t] = G\Sigma_tG' \tag{5.24}
\end{equation}

**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 \textbf{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

\[ u \sim N(\bar{u}, S) \quad \text{and} \quad v = a + Bu \implies v \sim N(a + B\bar{u}, BSB') \quad (5.25) \]

In particular, given our Gaussian assumptions on the primitives and the linearity of (5.18) we can see immediately that both \( x_t \) and \( y_t \) are Gaussian for all \( t \geq 0 \). \footnote{The correct way to argue this is by induction. Suppose that \( x_t \) is Gaussian. Then (5.18) and (5.25) 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.}

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 (5.21) and (5.22)

Letting \( \mu_t \) and \( \Sigma_t \) be as defined by these equations, we have

\[ x_t \sim N(\mu_t, \Sigma_t) \quad (5.26) \]

By similar reasoning combined with (5.23) and (5.24),

\[ y_t \sim N(G\mu_t, G\Sigma_t G') \quad (5.27) \]

**Ensemble Interpretations**

How should we interpret the distributions defined by (5.26)–(5.27)?

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 (5.19)

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 (5.27).

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 distributions 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 \( y^i_T \).
3. histogramming this sample

Just as the histogram approximates the population distribution, the *ensemble or cross-sectional average*

\[
\bar{y}_T := \frac{1}{I} \sum_{i=1}^{I} y^i_T
\]

approximates the expectation \( \mathbb{E}[y_T] = G\mu_T \) (as implied by the law of large numbers).

Here's a simulation comparing the ensemble averages and population means at time points \( t = 0, \ldots, 50 \).

The parameters are the same as for the preceding figures, and the sample size is relatively small (\( I = 20 \)).
The ensemble mean for $x_t$ is

$$\bar{x}_T := \frac{1}{I} \sum_{i=1}^{I} x^i_T \to \mu_T \quad (I \to \infty)$$

The limit $\mu_T$ 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

$$\frac{1}{I} \sum_{i=1}^{I} (x^i_T - \bar{x}_T)(x^i_T - \bar{x}_T)' \to \Sigma_T \quad (I \to \infty)$$

**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

\[ p(x, y) = p(y | x)p(x) \quad \text{(joint = conditional } \times \text{ marginal)} \]

From this rule we get \( p(x_0, x_1) = p(x_1 | x_0)p(x_0) \)

The Markov property \( p(x_t | x_{t-1}, \ldots, x_0) = p(x_t | x_{t-1}) \) and repeated applications of the preceding rule lead us to

\[ p(x_0, x_1, \ldots, x_T) = p(x_0) \prod_{t=0}^{T-1} p(x_{t+1} | x_t) \]

The marginal \( p(x_0) \) is just the primitive \( N(\mu_0, \Sigma_0) \)

In view of (5.18), the conditional densities are

\[ p(x_{t+1} | x_t) = N(Ax_t, CC') \]

**Autocovariance functions**

An important object related to the joint distribution is the *autocovariance function*

\[ \Sigma_{t+j,t} := \mathbb{E}[(x_{t+j} - \mu_{t+j})(x_t - \mu_t)'] \quad (5.28) \]

Elementary calculations show that

\[ \Sigma_{t+j,t} = A^j \Sigma_t \quad (5.29) \]

Notice that \( \Sigma_{t+j,t} \) in general depends on both \( j \), the gap between the two dates, and \( t \), the earlier date

### 5.4.4 Stationarity and Ergodicity

Stationarity and ergodicity are two properties that, when they hold, greatly aid analysis of linear state space models

Lets start with the intuition

**Visualizing Stability**

Lets 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*.

### Stationary Distributions

In our setting, a distribution $\psi_{\infty}$ is said to be *stationary* for $x_t$ if

$$x_t \sim \psi_{\infty} \quad \text{and} \quad x_{t+1} = Ax_t + Cw_{t+1} \implies x_{t+1} \sim \psi_{\infty}$$

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: $\psi_{\infty}$ is stationary for $x_t$ if

$$\psi_{\infty} = N(\mu_{\infty}, \Sigma_{\infty})$$

where $\mu_{\infty}$ and $\Sigma_{\infty}$ are fixed points of (5.21) and (5.22) respectively.

### 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 $\mu_\infty$ and $\Sigma_\infty$ as fixed points of (5.21) and (5.22) 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 (5.29), 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 $\mu_t$ and $\Sigma_t$ 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 $\mu_0, \Sigma_0, A, C$ assume values that imply that none of $\mu_t, \Sigma_t, \Sigma_{t+j,t}$ depends on $t$
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 $(\text{np.abs}(\text{np.linalg.eigvals}(A)) < 1).\text{all()} == \text{True}$

The difference equation (5.22) 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 $\mu_0$ and $\Sigma_0$

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 $x_t = [1 \quad y_t \quad y_{t-1}]'$

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

$$A = \begin{bmatrix} A_1 & a \\ 0 & 1 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} C_1 \\ 0 \end{bmatrix}$$

where

- $A_1$ is an $(n - 1) \times (n - 1)$ matrix
- $a$ is an $(n - 1) \times 1$ column vector

Let $x_t = [x'_{1t} \quad 1]'$ 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

$$\mu_{1,t+1} = A_1 \mu_{1,t} + a \quad \text{(5.30)}$$

Assume now that the moduli of the eigenvalues of $A_1$ are all strictly less than one
Then (5.30) has a unique stationary solution, namely,

$$\mu_{1\infty} = (I - A_1)^{-1}a$$

The stationary value of $\mu_t$ itself is then $\mu_{\infty} := [\mu'_{1,\infty} \quad 1]'$

The stationary values of $\Sigma_t$ and $\Sigma_{t+j,t}$ satisfy

$$\Sigma_{\infty} = A\Sigma_{\infty}A' + CC'$$
$$\Sigma_{t+j,t} = A^j\Sigma_{\infty}$$

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 (5.22) converge to the fixed point of the discrete Lyapunov equation in the first line of (5.31)

**Ergodicity**

Let's suppose that we were working with a covariance stationary process

In this case we know that the ensemble mean will converge to $\mu_{\infty}$ as the sample size $T$ 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

### 5.4.5 Noisy Observations

In some settings the observation equation \( y_t = G x_t \) 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

\[
\begin{align*}
x_{t+1} &= A x_t + C w_{t+1} \\
y_t &= G x_t + H v_t \\
x_0 &\sim N(\mu_0, \Sigma_0)
\end{align*}
\]

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 (5.23) and (5.24) now become

\[
\mathbb{E}[y_t] = \mathbb{E}[G x_t + H v_t] = G \mu_t
\]

(5.31)

The variance-covariance matrix of \( y_t \) is easily shown to be

\[
\text{Var}[y_t] = \text{Var}[G x_t + H v_t] = G \Sigma_t G' + HH'
\]

(5.32)

The distribution of \( y_t \) is therefore

\[
y_t \sim N(G \mu_t, G \Sigma_t G' + HH')
\]
5.4.6 Prediction

The theory of prediction for linear state space systems is elegant and simple

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

\[
E_t[x_{t+1}] := E[x_{t+1} | x_t, x_{t-1}, \ldots, x_0] = Ax_t
\]

The right-hand side follows from \( x_{t+1} = Ax_t + Cw_{t+1} \) and the fact that \( w_{t+1} \) is zero mean and independent of \( x_t, x_{t-1}, \ldots, x_0 \)

That \( E_t[x_{t+1}] = E[x_{t+1} | x_t] \) is an implication of \( \{x_t\} \) having the Markov property

The one-step-ahead forecast error is

\[
x_{t+1} - E_t[x_{t+1}] = Cw_{t+1}
\]

The covariance matrix of the forecast error is

\[
E[(x_{t+1} - E_t[x_{t+1}]) (x_{t+1} - E_t[x_{t+1}])'] = CC'
\]

More generally, we'd like to compute the \( j \)-step ahead forecasts \( E_t[x_{t+j}] \) and \( E_t[y_{t+j}] \)

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 \( E_t[w_{t+k}] = E[w_{t+k}] = 0 \)

It now follows from linearity of expectations that the \( j \)-step ahead forecast of \( x \) is

\[
E_t[x_{t+j}] = A^j x_t
\]

The \( j \)-step ahead forecast of \( y \) is therefore

\[
E_t[y_{t+j}] = E_t[Gx_{t+j} + Hv_{t+j}] = GA^j x_t
\]

Covariance of Prediction Errors

It is useful to obtain the covariance matrix of the vector of \( j \)-step-ahead prediction errors

\[
x_{t+j} - E_t[x_{t+j}] = \sum_{s=0}^{j-1} A^s C w_{t-s+j}
\] (5.33)
Evidently,

\[ V_j := \mathbb{E}_t[(x_t + j - \mathbb{E}_t[x_{t+j}]) (x_{t+j} - \mathbb{E}_t[x_{t+j}])'] = \sum_{k=0}^{j-1} A^k CC'A'^k \]  

(5.34)

\[ V_j \] defined in (5.34) can be calculated recursively via

\[ V_1 = CC' \] and

\[ V_j = CC' + AV_{j-1}A', \quad j \geq 2 \]  

(5.35)

\[ V_j \] is the conditional covariance matrix of the errors in forecasting \( x_{t+j} \), conditioned on time \( t \) information \( x_t \).

Under particular conditions, \( V_j \) converges to

\[ V_\infty = CC' + AV_\infty A' \]  

(5.36)

Equation (5.36) is an example of a discrete Lyapunov equation in the covariance matrix \( V_\infty \).

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.

**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 (5.18).

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

\[
\mathbb{E}_t \left[ \sum_{j=0}^{\infty} \beta^j y_{t+j} \right] = G[I + \beta A + \beta^2 A^2 + \cdots] x_t = G[I - \beta A]^{-1} x_t
\]

5.4.7 Code

Our preceding simulations and calculations are based on code in the file lss.py from the QuantEcon.py package. The code implements a class for handling linear state space models (simulations, calculating moments, etc.). One Python construct you might not be familiar with is the use of a generator function in the method moment_sequence(). Go back and read the relevant documentation if you've forgotten how generator functions work. Examples of usage are given in the solutions to the exercises.

5.4.8 Exercises

Exercise 1

Replicate this figure using the LinearStateSpace class from lss.py.

Exercise 2

Replicate this figure modulo randomness using the same class.

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.
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

5.4.9 Solutions

```python
import numpy as np
import matplotlib.pyplot as plt
from quantecon import LinearStateSpace

Exercise 1

_0, _1, _2 = 1.1, 0.8, -0.8
A = [[1, 0, 0],
     [1 -_0, _1, _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()
```
Exercise 2

\( _1, _2, _3, _4 = 0.5, -0.2, 0, 0.5 \)
\( \sigma = 0.2 \)

\( A = \begin{bmatrix} _1, & _2, & _3, & _4 \end{bmatrix}, \\
    \begin{bmatrix} 1, & 0, & 0, & 0 \end{bmatrix}, \\
    \begin{bmatrix} 0, & 1, & 0, & 0 \end{bmatrix}, \\
    \begin{bmatrix} 0, & 0, & 1, & 0 \end{bmatrix} \)

\( C = \begin{bmatrix} \sigma \end{bmatrix}, \\
    \begin{bmatrix} 0 \end{bmatrix}, \\
    \begin{bmatrix} 0 \end{bmatrix}, \\
    \begin{bmatrix} 0 \end{bmatrix} \)

\( G = \begin{bmatrix} 1, & 0, & 0, & 0 \end{bmatrix} \)

\( ar = \text{LinearStateSpace}(A, C, G, \mu_0=np\text{.ones}(4)) \)
\( x, y = ar\text{.simulate}(ts\_length=200) \)

\( fig, ax = \text{plt\text{.subplots}(figsize=(10, 6))} \)
\( y = y\text{.flatten()} \)
\( ax\text{.plot}(y, 'b-', lw=2, alpha=0.7) \)
\( ax\text{.grid()} \)
\( ax\text{.set_xxlabel('time')} \)
\( ax\text{.set_yylabel('$y_t$', fontsize=16)} \)
\( plt\text{.show()} \)
Exercise 3

```python
from scipy.stats import norm
import random

_1, _2, _3, _4 = 0.5, -0.2, 0, 0.5
σ = 0.1

A = [[_1, _2, _3, _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):
    _x, _y, _x, _y = next(m)
    population_means.append(float(_y))
ax.plot(population_means, color='g', lw=2, alpha=0.8, label='G\mu_t')
ax.legend(ncol=2)
plt.show()
\[ C = \begin{bmatrix} 1, & 0, & 0, & 0 \\ 0, & 1, & 0, & 0 \\ 0, & 0, & 1, & 0 \end{bmatrix} \]

\[ G = [1, 0, 0, 0] \]

\[ T_0 = 10 \]
\[ T_1 = 50 \]
\[ T_2 = 75 \]
\[ T_4 = 100 \]

```
ar = LinearStateSpace(A, C, G, mu_0=np.ones(4), Sigma_0=Sigma_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(('\(T\)', '\(T' + '1\)', '\(T' + '2\)'), fontsize=14)

\(\mu_x, \mu_y, \Sigma_x, \Sigma_y\) = ar.stationary_distributions()
ar.mu_0 = \mu_x
ar.Sigma_0 = \Sigma_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()
```
5.5 Finite Markov Chains

Contents

- Finite Markov Chains
  - Overview
  - Definitions
  - Simulation
  - Marginal Distributions
  - Irreducibility and Aperiodicity
  - Stationary Distributions
  - Ergodicity
  - Computing Expectations
  - Exercises
  - Solutions
5.5.1 Overview

Markov chains are one of the most useful classes of stochastic processes, being
- simple, flexible and supported by many elegant theoretical results
- valuable for building intuition about random dynamic models
- central to quantitative modeling in their own right

You will find them in many of the workhorse models of economics and finance

In this lecture we review some of the theory of Markov chains

We will also introduce some of the high quality routines for working with Markov chains available in QuantEcon.py

Prerequisite knowledge is basic probability and linear algebra

5.5.2 Definitions

The following concepts are fundamental

Stochastic Matrices

A stochastic matrix (or Markov matrix) is an $n \times n$ square matrix $P$ such that
1. each element of $P$ is nonnegative, and
2. each row of $P$ sums to one

Each row of $P$ can be regarded as a probability mass function over $n$ possible outcomes

It is too not difficult to check$^1$ that if $P$ is a stochastic matrix, then so is the $k$-th power $P^k$ for all $k \in \mathbb{N}$

Markov Chains

There is a close connection between stochastic matrices and Markov chains

To begin, let $S$ be a finite set with $n$ elements $\{x_1, \ldots, x_n\}$

The set $S$ is called the state space and $x_1, \ldots, x_n$ are the state values

A Markov chain $\{X_t\}$ on $S$ is a sequence of random variables on $S$ that have the Markov property

This means that, for any date $t$ and any state $y \in S$,

$$
P\{X_{t+1} = y \mid X_t\} = P\{X_{t+1} = y \mid X_t, X_{t-1}, \ldots\} \tag{5.37}
$$

In other words, knowing the current state is enough to know probabilities for future states

$^1$ Hint: First show that if $P$ and $Q$ are stochastic matrices then so is their product to check the row sums, try postmultiplying by a column vector of ones. Finally, argue that $P^n$ is a stochastic matrix using induction.
In particular, the dynamics of a Markov chain are fully determined by the set of values

\[ P(x, y) := P\{X_{t+1} = y \mid X_t = x\} \quad (x, y \in S) \quad (5.38) \]

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

\[ P_{ij} = P(x_i, x_j) \quad 1 \leq i, j \leq n \]

Going the other way, if we take a stochastic matrix \( P \), we can generate a Markov chain \( \{X_t\} \) as follows:

- draw \( X_0 \) from some specified distribution
- for each \( t = 0, 1, \ldots \), draw \( X_{t+1} \) from \( P(X_t, \cdot) \)

By construction, the resulting process satisfies (5.38)

**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

\[
P = \begin{pmatrix}
1 - \alpha & \alpha \\
\beta & 1 - \beta
\end{pmatrix}
\]

Once we have the values \( \alpha \) and \( \beta \), 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 cover such applications below

---

5.5. Finite Markov Chains
Example 2

Using US unemployment data, Hamilton \([Ham05]\) estimated the stochastic matrix

\[
P = \begin{pmatrix}
0.971 & 0.029 & 0 \\
0.145 & 0.778 & 0.077 \\
0 & 0.508 & 0.492
\end{pmatrix}
\]

where

- the frequency is monthly
- the first state represents normal growth
- the second state represents mild recession
- the third state represents severe recession

For example, the matrix tells us that when the state is normal growth, the state will again be normal growth next month with probability 0.97

In general, large values on the main diagonal indicate persistence in the process \(\{X_t\}\)

This Markov process can also be represented as a directed graph, with edges labeled by transition probabilities

\includegraphics[width=0.6\textwidth]{example_2.png}

Here ng is normal growth, mr is mild recession, etc.

5.5.3 Simulation

One natural way to answer questions about Markov chains is to simulate them

(To approximate the probability of event \(E\), we can simulate many times and count the fraction of times that \(E\) occurs)

Nice functionality for simulating Markov chains exists in \texttt{QuantEcon.py}

- Efficient, bundled with lots of other useful routines for handling Markov chains

However, its also a good exercise to roll our own routines \(\text{let's do that first and then come back to the methods in } \texttt{QuantEcon.py}\)

In these exercises well take the state space to be \(S = 0, \ldots, n - 1\)
Rolling our own

To simulate a Markov chain, we need its stochastic matrix $P$ and either an initial state or a probability distribution $\psi$ for 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 set to some fixed state or chosen from $\psi$
2. At each subsequent time $t$, the new state $X_{t+1}$ is drawn from $P(X_t, \cdot)$

In order to implement this simulation procedure, we need a method for generating draws from a discrete distributions.

For this task we'll use `DiscreteRV` from `QuantEcon`.

```python
import quantecon asqe
import numpy as np

ψ = (0.1, 0.9)  # Probabilities over sample space {0, 1}
cdf = np.cumsum(ψ)
qe.random.draw(cdf, 5)  # Generate 5 independent draws from ψ

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

Well write our code as a function that takes the following three arguments

- A stochastic matrix $P$
- An initial state $\text{init}$
- A positive integer $\text{sample_size}$ representing the length of the time series the function should return

```python
def mc_sample_path(P, init=0, sample_size=1000):
    # === make sure P is a NumPy array === #
    P = np.asarray(P)
    # === allocate memory === #
    X = np.empty(sample_size, dtype=int)
    X[0] = init
    # === convert each row of P into a distribution === #
    # In particular, P_dist[i] = the distribution corresponding to P[i, :]
    n = len(P)
    P_dist = [np.cumsum(P[i, :]) for i in range(n)]

    # === generate the sample path === #
    for t in range(sample_size - 1):
        X[t+1] = qe.random.draw(P_dist[X[t]])
    
    return X
```

Let's see how it works using the small matrix

$$P := \begin{pmatrix} 0.4 & 0.6 \\ 0.2 & 0.8 \end{pmatrix} \quad (5.39)$$

5.5. Finite Markov Chains
As well see later, for a long series drawn from \( P \), the fraction of the sample that takes value 0 will be about 0.25

If you run the following code you should get roughly that answer

\[
P = [[0.4, 0.6], [0.2, 0.8]]
\]

\[
X = mc_sample_path(P, sample_size=100000)
\]

\[
np.mean(X == 0)
\]

0.25128

Using QuantEcons 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

\[
P = [[0.4, 0.6], [0.2, 0.8]]
\]

\[
mc = qe.MarkovChain(P)
\]

\[
X = mc.simulate(ts_length=1000000)
\]

\[
np.mean(X == 0)
\]

0.250359

In fact the QuantEcon.py routine is *JIT compiled* and much faster

(Because its JIT compiled the first run takes a bit longer the function has to be compiled and stored in memory)

\[
\%timeit mc_sample_path(P, sample_size=1000000) # our version
\]

1 loops, best of 3: 6.86 s per loop

\[
\%timeit mc.simulate(ts_length=1000000) # qe version
\]

10 loops, best of 3: 72.5 ms per loop

Adding state values and initial conditions

If we wish to, we can provide a specification of state values to MarkovChain.

These state values can be integers, floats, or even strings.

The following code illustrates

\[
mc = qe.MarkovChain(P, state_values=('unemployed', 'employed'))
\]

\[
mc.simulate(ts_length=4, init='employed')
\]
array(['employed', 'unemployed', 'unemployed', 'employed'], dtype='<U10')

mc.simulate(ts_length=4, init='unemployed')

array(['unemployed', 'unemployed', 'unemployed', 'unemployed'], dtype='<U10')

mc.simulate(ts_length=4)  # Start at randomly chosen initial state

array(['unemployed', 'unemployed', 'unemployed', 'unemployed'], dtype='<U10')

If we want to simulate with output as indices rather than state values we can use

mc.simulate_indices(ts_length=4)

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

5.5.4 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 \( \psi_t \)

What then is the distribution of \( X_{t+1} \), or, more generally, of \( X_{t+m} \)?

**Solution**

Let \( \psi_t \) be the distribution of \( X_t \) for \( t = 0, 1, 2, \ldots \)

Our first aim is to find \( \psi_{t+1} \) given \( \psi_t \) 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:

\[
P\{X_{t+1} = y\} = \sum_{x \in S} P\{X_{t+1} = y \mid X_t = x\} \cdot P\{X_t = x\}
\]

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 $\psi_{t+1}$ and $\psi_t$ as row vectors (as is traditional in this literature), these $n$ equations are summarized by the matrix expression

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

(5.40)

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 (5.40), 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 $\psi_0$ 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

$$
X_0 \sim \psi_0 \implies X_m \sim \psi_0 P^m
$$

(5.41)

and, more generally,

$$
X_t \sim \psi_t \implies X_{t+m} \sim \psi_t P^m
$$

(5.42)

**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 (5.42), but now with $\psi_t$ putting all probability on state $x$

- 1 in the $x$-th position and zero elsewhere

Inserting this into (5.42), 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\} = P^m(x, y) = (x, y)$-th element of $P^m$

**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

\[ \psi P^6 \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \]

**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 (i.e., tending to infinite) population of workers, each of whose lifetime experiences are described by the specified dynamics, independently of one another.

Let \( \psi \) be the current *cross-sectional* distribution over \( \{0, 1\} \)

- For example, \( \psi(0) \) is the unemployment rate.

The cross-sectional distribution records the fractions of workers employed and unemployed at a given moment.

The same distribution also describes the fractions of a particular worker's career spent being employed and unemployed, respectively.

### 5.5.5 Irreducibility and Aperiodicity

Irreducibility and aperiodicity are central concepts of modern Markov chain theory.

Let's see what they're about.

**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:

```python
P = [[0.9, 0.1, 0],
     [0.4, 0.4, 0.2],
     [0.1, 0.1, 0.8]]
mc = qe.MarkovChain(P, ('poor', 'middle', 'rich'))
mc.is_irreducible
```

```
True
```

Here's a more pessimistic scenario, where the poor are poor forever.
This stochastic matrix is not irreducible, since, for example, \textit{rich} is not accessible from \textit{poor}

Let's confirm this

\[
P = \begin{bmatrix}
1.0 & 0.0 & 0.0 \\
0.1 & 0.8 & 0.1 \\
0.0 & 0.2 & 0.8 \\
\end{bmatrix}
\]

\[
mc = qe\text{.MarkovChain}(P, ('\textit{poor}', '\textit{middle}', '\textit{rich}'))
mc\text{.is_irreducible}
\]

\[
\text{False}
\]

We can also determine the communication classes

\[
mc\text{.communication_classes}
\]

\[
[\text{array(['poor'], dtype='<U6')}, \text{array(['middle', 'rich'], dtype='<U6'))}
\]

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.

Well come back to this a bit later.

\section*{Aperiodicity}

Loosely speaking, a Markov chain is called periodic if it cycles in a predictable way, and aperiodic otherwise.

Here's a trivial example with three states.

\[
\text{The chain cycles with period 3:}
\]

\[
P = \begin{bmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
1 & 0 & 0 \\
\end{bmatrix}
\]

\[
mc = qe\text{.MarkovChain}(P)
mc\text{.period}
\]
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, $D(x) = \{3, 6, 9, \ldots \}$ 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.

![Transition diagram](image)

We can confirm that the stochastic matrix is periodic as follows:

```python
P = [[0.0, 1.0, 0.0, 0.0],
     [0.5, 0.0, 0.5, 0.0],
     [0.0, 0.5, 0.0, 0.5],
     [0.0, 0.0, 1.0, 0.0]]
mc = qe.MarkovChain(P)
mc.period
2
mc.is_aperiodic
False
```

### 5.5.6 Stationary Distributions

As seen in (5.40), we can shift probabilities forward one unit of time via postmultiplication by $P$.

Some distributions are invariant under this updating process for example,

```python
P = np.array([ [.4, .6], [.2, .8]])
ψ = (0.25, 0.75)
ψ @ P
array([ 0.25, 0.75])
```

Such distributions are called **stationary**, or **invariant**.

Formally, a distribution $ψ^*$ on $S$ is called **stationary** for $P$ if $ψ^* = ψ^*P$.
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 well 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 a 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 $\psi^*$
2. For any initial distribution $\psi_0$, we have $\|\psi_0 P^t - \psi^*\| \to 0$ as $t \to \infty$

For a proof, see, for example, theorem 5.2 of [Haggstrom02]

(Note that part 1 of the theorem 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

**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 $\alpha \to 0$.

**Calculating Stationary Distributions**

As discussed above, a given Markov matrix $P$ can have many stationary distributions.

That is, there can be many row vectors $\psi$ such that $\psi = \psi P$.

In fact if $P$ has two distinct stationary distributions $\psi_1, \psi_2$ 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 $\psi$, 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.

```python
P = [[0.4, 0.6], [0.2, 0.8]]
mc = qe.MarkovChain(P)
mc.stationary_distributions  # Show all stationary distributions
```

The stationary distribution is unique.

**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 $\psi^*$ as a stochastic steady state.

The convergence in the theorem is illustrated in the next figure.

```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

P = ((0.971, 0.029, 0.000),
     (0.145, 0.778, 0.077),
     (0.000, 0.508, 0.492))
P = np.array(P)

psi = (0.0, 0.2, 0.8)  # Initial condition
```
Here

- $P$ is the stochastic matrix for recession and growth considered above
- The highest red dot is an arbitrarily chosen initial probability distribution $\psi$, represented as a vector in $\mathbb{R}^3$
- The other red dots are the distributions $\psi P^t$ for $t = 1, 2, \ldots$
The black dot is $\psi^*$

The code for the figure can be found here you might like to try experimenting with different initial conditions

### 5.5.7 Ergodicity

Under irreducibility, yet another important result obtains: For all $x \in S$,

$$\frac{1}{n} \sum_{t=1}^{m} 1\{ X_t = x \} \rightarrow \psi^*(x) \quad \text{as } m \rightarrow \infty$$

(5.43)

Here

- $1\{ X_t = x \} = 1$ 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 (5.43) is valid.

The convergence in (5.43) is a special case of a law of large numbers result for Markov chains see EDTC, section 4.3.4 for some additional information

### 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.
5.5.8 Computing Expectations

We are interested in computing expectations of the form

\[ E[h(X_t)] \]  \hspace{1cm} (5.44) 

and conditional expectations such as

\[ E[h(X_{t+k}) \mid X_t = x] \]  \hspace{1cm} (5.45) 

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, well think of as the column vector

\[
h = \begin{pmatrix} h(x_1) \\ \vdots \\ h(x_n) \end{pmatrix}
\]

The unconditional expectation (5.44) is easy: We just sum over the distribution of \( X_t \) to get

\[ E[h(X_t)] = \sum_{x \in S} (\psi P^t)(x)h(x) \]

Here \( \psi \) is the distribution of \( X_0 \)

Since \( \psi \) and hence \( \psi P^t \) are row vectors, we can also write this as

\[ E[h(X_t)] = \psi P^t h \]

For the conditional expectation (5.45), 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

\[ E[h(X_{t+k}) \mid X_t = x] = (P^k h)(x) \]  \hspace{1cm} (5.46) 

The vector \( P^k h \) stores the conditional expectation \( E[h(X_{t+k}) \mid X_t = x] \) over all \( x \)

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

\[
E \left[ \sum_{j=0}^{\infty} \beta^j h(X_{t+j}) \mid X_t = x \right] = [(I - \beta P)^{-1} h](x)
\]
where

\[(I - \beta P)^{-1} = I + \beta P + \beta^2 P^2 + \cdots\]

Premultiplication by \((I - \beta P)^{-1}\) amounts to applying the \textit{resolvent operator}

**5.5.9 Exercises**

**Exercise 1**

According to the discussion \textit{above}, if a workers 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

\[\bar{X}_m := \frac{1}{m} \sum_{t=1}^{m} 1\{X_t = 0\}\]

Your exercise is to illustrate this convergence

First,

- generate one simulated time series \(\{X_t\}\) of length 10,000, starting at \(X_0 = 0\)
- plot \(\bar{X}_m - p\) against \(m\), where \(p\) is as defined above

Second, repeat the first step, but this time taking \(X_0 = 1\)

In both cases, set \(\alpha = \beta = 0.1\)

The result should look something like the following modulo randomness, of course
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 [DLP13].)

To understand the issue, consider the set of results returned by a query to a web search engine.

For the user, it is desirable to

1. receive a large set of accurate matches
2. have the matches returned in order, where the order corresponds to some measure of importance

Ranking according to a measure of importance is the problem we now consider.

The methodology developed to solve this problem by Google founders Larry Page and Sergey Brin is known as PageRank.

To illustrate the idea, consider the following diagram.
Imagine that this is a miniature version of the WWW, with

- each node representing a web page
- each arrow representing the existence of a link from one page to another

Now let's think about which pages are likely to be important, in the sense of being valuable to a search engine user.

One possible criterion for 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

- \( \ell_i \) 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 $P(i, j) = \mathbf{1}(i \rightarrow j)/\ell_i$ where $\mathbf{1}(i \rightarrow j) = 1$ 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

$$r_j = \sum_{i \in L_j} \frac{r_i}{\ell_i} = \sum_{i \in L_j} \mathbf{1}(i \rightarrow j) \frac{r_i}{\ell_i} = \sum_{i \in L_j} P(i, j) r_i$$

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.

The data for this graph is in the web_graph_data.txt file. You can also view it here.

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 $h$.

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.

```python
import re

re.findall(r'\w', 'x +++ y +++++ z')  # \w matches alphanumerics

['x', 'y', 'z']

re.findall(r'\w', 'a ^ b && $ c')
```

5.5. Finite Markov Chains 437
['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 \( a \).

**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 \([\text{Tau86}]\) is the most common method for approximating this continuous state process with a finite state Markov chain. A routine for this already exists in `QuantEcon.py` but let's write our own version as an exercise.

As a first step we choose

- \( n \), the number of states for the discrete approximation
- \( m \), an integer that parameterizes the width of the state space

Next we create a state space \( \{x_0, \ldots, x_{n-1}\} \subset \mathbb{R} \) and a stochastic \( n \times n \) matrix \( P \) such that

- \( x_0 = -m \sigma_y \)
- \( x_{n-1} = m \sigma_y \)
- \( x_{i+1} = x_i + s \) where \( s = (x_{n-1} - x_0)/(n - 1) \)

Let \( F \) be the cumulative distribution function of the normal distribution \( N(0, \sigma_y^2) \). The values \( P(x_i, x_j) \) are computed to approximate the AR(1) process omitting the derivation, the rules are as follows:

1. If \( j = 0 \), then set
   \[ P(x_i, x_j) = P(x_i, x_0) = F(x_0 - \rho x_i + s/2) \]

2. If \( j = n - 1 \), then set
   \[ P(x_i, x_j) = P(x_i, x_{n-1}) = 1 - F(x_{n-1} - \rho x_i - s/2) \]

3. Otherwise, set
\[ P(x_i, x_j) = F(x_j - \rho x_i + s/2) - F(x_j - \rho x_i - s/2) \]

The exercise is to write a function `approx_markov(rho, sigma_u, m=3, n=7)` that returns \{x_0, \ldots, x_{n-1}\} \subset \mathbb{R} and an \( n \times n \) matrix \( P \) as described above

- Even better, write a function that returns an instance of `QuantEcon.pys MarkovChain` class

### 5.5.10 Solutions

```python
import numpy as np
import matplotlib.pyplot as plt
from quantecon import MarkovChain

Exercise 1

Compute the fraction of time that the worker spends unemployed, and compare it to the stationary probability

\[ \alpha = \beta = 0.1 \]
\[ N = 10000 \]
\[ p = \beta / (\alpha + \beta) \]

\[ P = \begin{pmatrix} 1 - \alpha & \alpha \\ \beta & 1 - \beta \end{pmatrix} \]
\[ P = \text{np.array}(P) \]
\[ mc = \text{MarkovChain}(P) \]

```python
fig, ax = plt.subplots(figsize=(9, 6))
ax.set_ymlim(-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='$X_0 = \{x_0\}$')
    ax.plot(X_bar - p, 'k-', alpha=0.6)  # Overlay in black--make lines clearer
ax.legend(loc='upper right')
plt.show()
```
Exercise 2

First save the data into a file called `web_graph_data.txt` by executing the next cell

```python
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;
```
Overwriting web_graph_data.txt

```
""
Return list of pages, ordered by rank
""
import numpy as np
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):
```

5.5. Finite Markov Chains

441
print(f'{name} : {rank:.4}')

Rankings
***
g: 0.1607
j: 0.1594
m: 0.1195
n: 0.1088
k: 0.09106
b: 0.08326
e: 0.05312
i: 0.05312
c: 0.04834
h: 0.0456
l: 0.03202
d: 0.03056
f: 0.01164
a: 0.002911

Exercise 3

A solution from the QuantEcon.py library can be found here

5.6 Continuous State Markov Chains

Contents

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

5.6.1 Overview

In a previous lecture we learned about finite Markov chains, a relatively elementary class of stochastic dynamic models
The present lecture extends this analysis to continuous (i.e., uncountable) state Markov chains.

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

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

- evolve in discrete time
- are often nonlinear

The fact that we accommodate nonlinear models here is significant, because linear stochastic models have their own highly developed tool set, as well 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 is applicable to the model?
  - If so, how can we compute it?

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

**Note:** For some people, the term Markov chain always refers to a process with a finite or discrete state space. We follow the mainstream mathematical literature (e.g., [MT09]) in using the term to refer to any discrete time Markov process.

### 5.6.2 The Density Case

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

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

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

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

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

**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 $P = P[i, j]$ 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,

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

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 $P[\cdot, i]$)

In this section, we will 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 $p : S \times S \to \mathbb{R}$ 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

$$p_w(x, y) := \frac{1}{\sqrt{2\pi}} \exp \left\{ -\frac{(y - x)^2}{2} \right\} \quad (5.47)$$

What kind of model does $p_w$ represent?

The answer is, the (normally distributed) random walk

$$X_{t+1} = X_t + \xi_{t+1} \quad \text{where} \quad \{\xi_t\} \overset{\text{iid}}{\sim} N(0, 1) \quad (5.48)$$

To see this, let's find the stochastic kernel $p$ corresponding to (5.48)

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

Letting $X_t = x$ in (5.48) 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 (5.47)
Connection to Stochastic Difference Equations

In the previous section, we made the connection between stochastic difference equation (5.48) and stochastic kernel (5.47)

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

\[ X_{t+1} = \mu(X_t) + \sigma(X_t) \xi_{t+1} \]  
(5.49)

Here we assume that

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

**Example 1:** The random walk (5.48) is a special case of (5.49), with \( \mu(x) = x \) and \( \sigma(x) = 1 \)

**Example 2:** Consider the ARCH model

\[ X_{t+1} = \alpha X_t + \sigma_t \xi_{t+1}, \quad \sigma_t^2 = \beta + \gamma X_t^2, \quad \beta, \gamma > 0 \]

Alternatively, we can write the model as

\[ X_{t+1} = \alpha X_t + (\beta + \gamma X_t^2)^{1/2} \xi_{t+1} \]  
(5.50)

This is a special case of (5.49) 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

\[ k_{t+1} = sA_{t+1} f(k_t) + (1 - \delta)k_t \]  
(5.51)

Here

- \( s \) is the rate of savings
- \( A_{t+1} \) is a production shock
  - The \( t + 1 \) subscript indicates that \( A_{t+1} \) is not visible at time \( t \)
- \( \delta \) is a depreciation rate
- \( f: \mathbb{R}_+ \to \mathbb{R}_+ \) is a production function satisfying \( f(k) > 0 \) whenever \( k > 0 \)
(The fixed savings rate can be rationalized as the optimal policy for a particular set of technologies and preferences (see [LS18], section 3.1.2), although we omit the details here)

Equation (5.51) is a special case of (5.49) with \( \mu(x) = (1 - \delta)x \) and \( \sigma(x) = sf(x) \)

Now let's obtain the stochastic kernel corresponding to the generic model (5.49)

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

\[
f_V(v) = \frac{1}{b} f_U \left( \frac{v - a}{b} \right)
\]

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

Taking (5.52) as given for the moment, we can obtain the stochastic kernel \( p \) for (5.49) 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 (5.52),

\[
p(x, y) = \frac{1}{\sigma(x)} \phi \left( \frac{y - \mu(x)}{\sigma(x)} \right)
\]

For example, the growth model in (5.51) has stochastic kernel

\[
p(x, y) = \frac{1}{sf(x)} \phi \left( \frac{y - (1 - \delta)x}{sf(x)} \right)
\]

where \( \phi \) is the density of \( A_{t+1} \)

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

### Distribution Dynamics

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

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

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

Letting \( \psi_{t+1} \) denote the distribution of \( X_{t+1} \), the answer we gave was that

\[
\psi_{t+1}[j] = \sum_{i \in S} P[i, j] \psi_t[i]
\]
This intuitive equality states that the probability of being at $j$ tomorrow is the probability of visiting $i$ today and then going on to $j$, summed over all possible $i$.

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

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

(5.55)

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

$$
(\psi P)(y) = \int p(x, y)\psi(x) \, dx
$$

(5.56)

This operator is usually called the Markov operator corresponding to $p$.

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

With this notation, we can write (5.55) more succinctly as $\psi_{t+1}(y) = (\psi_t P)(y)$ for all $y$, or, dropping the $y$ and letting $\equiv$ indicate equality of functions,

$$
\psi_{t+1} \equiv \psi_t P
$$

(5.57)

Equation (5.57) tells us that if we specify a distribution for $\psi_0$, then the entire sequence of future distributions can be obtained by iterating with $P$.

It's interesting to note that (5.57) is a deterministic difference equation.

Thus, by converting a stochastic difference equation such as (5.49) 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.
Computation

To learn about the dynamics of a given process, it is 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 (5.56) and (5.57) using numerical integration.

However, to produce \( \psi P \) from \( \psi \) via (5.56), 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:

\[
{k_{t+1} = sA_{t+1}f(k_t) + (1 - \delta)k_t}
\]  

(5.58)

Our aim is to compute the sequence \( \{\psi_t\} \) associated with this model and fixed initial condition \( \psi_0 \).

To approximate \( \psi_t \) by simulation, recall that, by definition, \( \psi_t \) is the density of \( k_t \) given \( k_0 \sim \psi_0 \).

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

1. draw \( k_0 \) from the specified initial condition \( \psi_0 \)
2. draw the shocks \( A_1, \ldots, A_t \) from their specified density \( \phi \)
3. compute \( k_t \) iteratively via (5.58)

If we repeat this \( n \) times, we get \( n \) independent observations \( k^n_1, \ldots, k^n_t \).

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

A naive approach would be to use a histogram, or perhaps a *smoothed histogram* using SciPys `gaussian_kde` function.

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

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

Now we take these \( n \) observations \( k^n_{t-1}, \ldots, k^n_{t-1} \) and form the estimate

\[
\psi^n_t(y) = \frac{1}{n} \sum_{i=1}^{n} p(k^n_{t-1}, y)
\]  

(5.59)

where \( p \) is the growth model stochastic kernel in (5.54).

What is the justification for this slightly surprising estimator?
The idea is that, by the strong law of large numbers,

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

with probability one as \( n \to \infty \)

Here the first equality is by the definition of \( \psi_{t-1} \), and the second is by (5.55)

We have just shown that our estimator \( \psi_t^p(y) \) in (5.59) 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)

**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 (5.59) 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

**Example**

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

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

# == 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
   = lognorm(a_σ)

def p(x, y):
```

5.6. Continuous State Markov Chains
Stochastic kernel for the growth model with Cobb-Douglas production. Both x and y must be strictly positive.

d = s * x**alpha

return pdf((y - (1 - delta) * 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 = .rvs((n, T))
k[:, 0] = psi_0.rvs(n)  # Draw first column from initial distribution
for t in range(T-1):
    k[:, t+1] = s * A[:, t] * k[:, t]**alpha + (1 - delta) * 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 psi, g in zip(laes, greys):
    ax.plot(ygrid, psi(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).

### 5.6.3 Beyond Densities

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

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

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

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

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

To illustrate the issues, recall that Hopenhayn and Rogerson [HR93] study a model of firm dynamics where individual firm productivity follows the exogenous process

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

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 end points 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

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

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

The distribution dynamics in (5.55) are then replaced by

\[ F_{t+1}(y) = \int G(x, y) F_t(dx) \quad (5.60) \]

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

The intuition behind (5.60) is essentially the same as for (5.55).

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.
5.6.4 Stability

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

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

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

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

Theoretical Results

Analogous to the finite case, given a stochastic kernel \( p \) and corresponding Markov operator as defined in (5.56), a density \( \psi^* \) on \( S \) is called stationary for \( P \) if it is a fixed point of the operator \( P \).

In other words,

\[
\psi^*(y) = \int p(x, y)\psi^*(x) \, dx, \quad \forall y \in S
\]  
(5.61)

As with the finite case, if \( \psi^* \) is stationary for \( P \), and the distribution of \( X_0 \) is \( \psi^* \), then, in view of (5.57), \( X_t \) will have this same distribution for all \( t \).

Hence \( \psi^* \) is the stochastic equivalent of a steady state.

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

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

Even existence can fail very easily.

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

However, there are well-known conditions under which a stationary density \( \psi^* \) exists.

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

\[
\forall \psi \in \mathcal{D}, \quad \psi P^t \to \psi^* \quad \text{as} \quad t \to \infty
\]  
(5.62)

This combination of existence, uniqueness and global convergence in the sense of (5.62) is often referred to as global stability.

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

\[
\frac{1}{n} \sum_{i=1}^{n} h(X_t) \to \int h(x)\psi^*(x) \, dx \quad \text{as} \quad n \to \infty
\]  
(5.63)

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

5.6. Continuous State Markov Chains
Note that the convergence in (5.63) does not depend on the distribution (or value) of \(X_0\) This is actually very important for simulation: it means we can learn about \(\psi^*\) (i.e., approximate the right hand side of (5.63) via the left hand side) without requiring any special knowledge about what to do with \(X_0\)

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

In essence, it must be the case that

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

For one such set of conditions see theorem 8.2.14 of EDTC

In addition

- \[SLP89\] contains a classic (but slightly outdated) treatment of these topics
- From the mathematical literature, \[LM94\] and \[MT09\] give outstanding in depth treatments
- Section 8.1.2 of EDTC provides detailed intuition, and section 8.3 gives additional references
- EDTC, section 11.3.4 provides a specific treatment for the growth model we considered in this lecture

**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 (5.62) 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.

**Computing Stationary Densities**

In the preceding figure, each sequence of densities is converging towards the unique stationary density \( \psi^* \). Even from this figure we can get a fair idea what \( \psi^* \) looks like, and where its mass is located.

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

Let's say that we have a model of the form (5.49) that is stable and ergodic.

Let \( p \) be the corresponding stochastic kernel, as given in (5.53).

To approximate the stationary density \( \psi^* \), we can simply generate a long time series \( X_0, X_1, \ldots, X_n \) and estimate \( \psi^* \) via...
\[ \psi_n^*(y) = \frac{1}{n} \sum_{t=1}^{n} p(X_t, y) \]  

(5.64)

This is essentially the same as the look ahead estimator (5.59), except that now the observations we generate are a single time series, rather than a cross section.

The justification for (5.64) 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 (5.63) and the equality on the right is by (5.61).

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.

### 5.6.5 Exercises

#### Exercise 1

Consider the simple threshold autoregressive model

\[ X_{t+1} = \theta |X_t| + (1 - \theta^2)^{1/2} \xi_{t+1} \quad \text{where} \quad \{\xi_t\} \sim \text{IID } N(0, 1) \]

(5.65)

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

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

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

(5.66)

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

As an exercise, compute the look ahead estimate of \( \psi^* \), as defined in (5.64), and compare it with \( \psi^* \) in (5.66) 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

**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 in `stochasticgrowth.py`

Use the same parameters

For the four initial distributions, use the shifted beta distributions

\[ \psi_0 = \text{beta}(5, 5, \text{scale}=0.5, \text{loc}=1+2) \]
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.

```python
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()
```

The three data sets are

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

The figure looks as follows.

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 (5.65)

We know that the distribution of $X_t$ will converge to (5.66) 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

```python
initial_conditions = np.linspace(δ, 0, J)
```

For each $X_0$ in this set,

1. Generate $k$ time series of length $n$, each starting at $X_0$ and obeying (5.65)
2. Create a boxplot representing $n$ distributions, where the $t$-th distribution shows the $k$ observations of $X_t$

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

### 5.6.6 Solutions

#### 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.

```python
from scipy.stats import norm, gaussian_kde

= 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 norm.pdf((y - θ * np.abs(x)) / d) / d

Z = rvs(n)
```
\[ X = \text{np.empty}(n) \]
\[
\text{for } t \text{ in range}(n-1):
    X[t+1] = \theta \ast \text{np.abs}(X[t]) + d \ast Z[t]
\]
\[
\psi_{\text{est}} = \text{LAE}(p, X)
\]
\[
k_{\text{est}} = \text{gaussian_kde}(X)
\]

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

**Exercise 2**

Here's one program that does the job

```python
# == Define parameters == #
s = 0.2
\delta = 0.1
a_\sigma = 0.4
\alpha = 0.4

# A = \exp(B) \text{ where } B \sim N(0, a_\sigma)
# f(k) = k^{4}\alpha
```
= lognorm(a_σ)

```python
def p(x, y):
    """Stochastic kernel, vectorized in x. Both x and y must be positive."""
    d = s * x**α
    return .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 = .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()
```

5.6. Continuous State Markov Chains 461
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.

```python
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])')
```

462 Chapter 5. Tools and Techniques
```python
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()
```
5.6.7 Appendix

Here’s the proof of (5.52)

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 (5.52)

5.7 A First Look at the Kalman Filter

### Contents

- A First Look at the Kalman Filter
  - Overview
  - The Basic Idea
  - Convergence
  - Implementation
  - Exercises
  - Solutions

5.7.1 Overview

This lecture provides a simple and intuitive introduction to the Kalman filter, for those who either

- have heard of the Kalman filter but don’t know how it works, or
- know the Kalman filter equations, but don’t know where they come from

For additional (more advanced) reading on the Kalman filter, see

- [LS18], section 2.7.
- [AM05]

The second reference presents a comprehensive treatment of the Kalman filter

Required knowledge: Familiarity with matrix manipulations, multivariate normal distributions, covariance matrices, etc.

5.7.2 The Basic Idea

The Kalman filter has many applications in economics, but for now let’s pretend that we are rocket scientists
A missile has been launched from country Y and our mission is to track it. Let \( x \in \mathbb{R}^2 \) denote the current location of the missile—a pair indicating latitude-longitude coordinates on a map.

At the present moment in time, the precise location \( x \) is unknown, but we do have some beliefs about \( x \). One way to summarize our knowledge is a point prediction \( \hat{x} \),

- But what if the President wants to know the probability that the missile is currently over the Sea of Japan?
- Then it is better to summarize our initial beliefs with a bivariate probability density \( p \)
  
  \[ \int_E p(x) dx \]

  indicates the probability that we attach to the missile being in region \( E \).

The density \( p \) is called our prior for the random variable \( x \).

To keep things tractable in our example, we assume that our prior is Gaussian. In particular, we take

\[
p = N(\hat{x}, \Sigma) \tag{5.67}
\]

where \( \hat{x} \) is the mean of the distribution and \( \Sigma \) is a \( 2 \times 2 \) covariance matrix. In our simulations, we will suppose that

\[
\hat{x} = \begin{pmatrix} 0.2 \\ -0.2 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} 0.4 & 0.3 \\ 0.3 & 0.45 \end{pmatrix} \tag{5.68}
\]

This density \( p(x) \) is shown below as a contour map, with the center of the red ellipse being equal to \( \hat{x} \).

```python
from scipy import linalg
import numpy as np
import matplotlib.cm as cm
from matplotlib.mlab import bivariate_normal
import matplotlib.pyplot as plt

# == Set up the Gaussian prior density p == #
Sigma = np.matrix([[0.4, 0.3], [0.3, 0.45]])
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 = np.matrix([[1, 0], [0, 1]])
G = np.matrix(G)
R = 0.5 * Sigma
# == The matrices A and Q == #
A = np.matrix([[1.2, 0], [0, -0.2]])
A = np.matrix(A)
Q = 0.3 * Sigma
# == 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 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()
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 $y$

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
\[ y = Gx + v, \quad \text{where} \quad v \sim N(0, R) \]  

(5.69)

Here \( G \) and \( R \) are 2 \times 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 (5.69), the conditional density \( p(y | x) \) is \( N(Gx, R) \)
- \( p(y) \) does not depend on \( x \), and enters into the calculations only as a normalizing constant

Because we are in a linear and Gaussian framework, the updated density can be computed by calculating population linear regressions

In particular, the solution is known\(^1\) to be

\[
p(x | y) = N(\hat{x}^F, \Sigma^F)
\]

where

\[
\hat{x}^F := \hat{x} + \Sigma G'(G\Sigma G' + R)^{-1}(y - G\hat{x}) \quad \text{and} \quad \Sigma^F := \Sigma - \Sigma G'(G\Sigma G' + R)^{-1}G\Sigma
\]

(5.70)

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

\[^1\] See, for example, page 93 of [Bis06]. To get from his expressions to the ones used above, you will also need to apply the Woodbury matrix identity.
Our new density twists the prior $p(x)$ in a direction determined by the new information $y - G\hat{x}$

In generating the figure, we set $G$ to the identity matrix and $R = 0.5\Sigma$ for $\Sigma$ defined in (5.68)

**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,

\[ x_{t+1} = Ax_t + w_{t+1}, \quad \text{where} \quad w_t \sim N(0, Q) \quad (5.71) \]

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 (5.71), 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 (5.70) tell us that

\[
E[Ax^F + w] = AEX^F + EW = A\hat{x}^F + A\Sigma G'(G\Sigma G' + R)^{-1}(y - G\hat{x})
\]

and

\[
\text{Var}[Ax^F + w] = A\text{Var}[x^F]A' + Q = A\Sigma^F A' + Q = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q
\]

The matrix \( A\Sigma G'(G\Sigma G' + R)^{-1} \) is often written as \( K_{\Sigma} \) and called the Kalman gain.

- The subscript \( \Sigma \) has been added to remind us that \( K_{\Sigma} \) depends on \( \Sigma \), but not \( y \) or \( \hat{x} \).

Using this notation, we can summarize our results as follows.

Our updated prediction is the density \( N(\hat{x}_{\text{new}}, \Sigma_{\text{new}}) \) where

\[
\hat{x}_{\text{new}} := A\hat{x} + K_{\Sigma}(y - G\hat{x})
\]

\[
\Sigma_{\text{new}} := A\Sigma A' - K_{\Sigma}G\Sigma A' + Q
\]

- The density \( p_{\text{new}}(x) = N(\hat{x}_{\text{new}}, \Sigma_{\text{new}}) \) is called the predictive distribution.

The predictive distribution is the new density shown in the following figure, where the update has used parameters

\[
A = \begin{pmatrix} 1.2 & 0.0 \\ 0.0 & -0.2 \end{pmatrix}, \quad Q = 0.3 \times \Sigma
\]
import numpy as np
cimport matplotlib.pyplot as plt

from QuantEcon.lecturespython3 import random_generator

# Density 2
M = G.T @ linalg.inv(G @ G.T + R)
x_hat_F = x_hat + M @ (y - G @ x_hat)
Sigma_F = Sigma - M @ G @ Sigma
Z_F = gen_gaussian_plot_vals(x_hat_F, Sigma_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_Sigma = A @ Sigma_F @ A.T + Q
new_Z = gen_gaussian_plot_vals(new_x_hat, new_Sigma)
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()
The Recursive Procedure

Lets look back at what weve 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 \((5.71)\) 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 \((5.69)\)
4. Compute the predictive distribution \( p_{t+1}(x) = N(\hat{x}_{t+1}, \Sigma_{t+1}) \) from the filtering distribution and \((5.71)\)
5. Increment \( t \) by one and go to step 1

Repeating \((5.72)\), the dynamics for \( \hat{x}_t \) and \( \Sigma_t \) are as follows

\[
\begin{align*}
\hat{x}_{t+1} &= A\hat{x}_t + K_{\Sigma_t}(y_t - G\hat{x}_t) \\
\Sigma_{t+1} &= A\Sigma_t A' - K_{\Sigma_t}G\Sigma_t A' + Q
\end{align*}
\]

These are the standard dynamic equations for the Kalman filter (see, for example, \([LS18]\), page 58)

5.7.3 Convergence

The matrix \( \Sigma_t \) 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 dont), the transition equation \((5.71)\) 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 \( \Sigma_t \) converges to a constant matrix as \( t \to \infty \)

To study this topic, lets expand the second equation in \((5.72)\):

\[
\Sigma_{t+1} = A\Sigma_t A' - A\Sigma_t G'(G\Sigma_t G' + R)^{-1}G\Sigma_t A' + Q \tag{5.72}
\]
This is a nonlinear difference equation in $\Sigma_t$

A fixed point of (5.72) is a constant matrix $\Sigma$ such that

$$\Sigma = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q$$

Equation (5.72) is known as a discrete time Riccati difference equation

Equation (5.73) is known as a discrete time algebraic Riccati equation

Conditions under which a fixed point exists and the sequence $\{\Sigma_t\}$ converges to it are discussed in [AHMS96] and [AM05], chapter 4

A sufficient (but not necessary) condition is that all the eigenvalues $\lambda_i$ of $A$ satisfy $|\lambda_i| < 1$ (cf. e.g., [AM05], p. 77)

(This strong condition assures that the unconditional distribution of $x_t$ converges as $t \to +\infty$)

In this case, for any initial choice of $\Sigma_0$ that is both nonnegative and symmetric, the sequence $\{\Sigma_t\}$ in (5.72) converges to a nonnegative symmetric matrix $\Sigma$ that solves (5.73)

## 5.7.4 Implementation

The class Kalman from the QuantEcon.py package implements the Kalman filter

- Instance data consists of:
  - the moments $(\hat{x}_t, \Sigma_t)$ of the current prior
  - An instance of the LinearStateSpace class from QuantEcon.py

The latter represents a linear state space model of the form

$$x_{t+1} = Ax_t + Cw_{t+1}$$
$$y_t = Gx_t + Hv_t$$

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 (5.73) and the corresponding (stationary) Kalman gain

You can view the program on GitHub

### 5.7.5 Exercises

**Exercise 1**

Consider the following simple application of the Kalman filter, loosely based on [LS18], section 2.9.2

Suppose that

- all variables are scalars
- the hidden state \( \{x_t\} \) is in fact constant, equal to some \( \theta \in \mathbb{R} \) unknown to the modeler

State dynamics are therefore given by (5.71) with \( A = 1, Q = 0 \) and \( x_0 = \theta \)

The measurement equation is \( y_t = \theta + v_t \) where \( v_t \) is \( N(0,1) \) and iid

The task of this exercise to simulate the model and, using the code from kalman.py, plot the first five predictive densities \( p_t(x) = N(\hat{x}_t, \Sigma_t) \)

As shown in [LS18], sections 2.9.1–2.9.2, these distributions asymptotically put all mass on the unknown value \( \theta \)

In the simulation, take \( \theta = 10, \hat{x}_0 = 8 \) and \( \Sigma_0 = 1 \)

Your figure should – modulo randomness – look something like this
Exercise 2

The preceding figure gives some support to the idea that probability mass converges to $\theta$.

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.
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 \( E[x_t \mid x_{t-1}] \), which in this case is \( Ax_{t-1} \)

The conditional expectation is known to be the optimal prediction method in terms of minimizing mean squared error

(More precisely, the minimizer of \( E \|x_t - g(x_{t-1})\|^2 \) with respect to \( g \) is \( g^*(x_{t-1}) := E[x_t \mid x_{t-1}] \))

Thus we are comparing the Kalman filter against a competitor who has more information (in the sense of being able to observe the latent state) and behaves optimally in terms of minimizing squared error

Our horse race will be assessed in terms of 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 \( 2 \times 2 \) 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

**Exercise 4**

Try varying the coefficient 0.3 in \( Q = 0.3I \) up and down

Observe how the diagonal values in the stationary solution \( \Sigma \) (see (5.73)) 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.

### 5.7.6 Solutions

```python
from quantecon import Kalman
from quantecon import LinearStateSpace
from scipy.stats import norm

Exercise 1

```python
# == 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()
```
from scipy.integrate import quad

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

Exercise 2
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 -, θ +)
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()
# === 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 ===#
Sigma = [[0.9, 0.3], [0.3, 0.9]]
Sigma = np.array(Sigma)
x_hat = np.array([8, 8])

# === Initialize the Kalman filter ===#
kn = Kalman(ss, x_hat, Sigma)

# == 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()
5.7. A First Look at the Kalman Filter
This section of the course contains foundational models for dynamic economic modeling. Most are single agent problems that take the activities of other agents as given. Later we will look at full equilibrium problems.

6.1 Shortest Paths

Contents

- Shortest Paths
  - Overview
  - Outline of the Problem
  - Finding Least-Cost Paths
  - Solving for $J$
  - Exercises
  - Solutions

6.1.1 Overview

The shortest path problem is a classic problem in mathematics and computer science with applications in

- Economics (sequential decision making, analysis of social networks, etc.)
- Operations research and transportation
- Robotics and artificial intelligence
- Telecommunication network design and routing
- etc., etc.

Variations of the methods we discuss in this lecture are used millions of times every day, in applications such as
• Google Maps
• routing packets on the internet

For us, the shortest path problem also provides a nice introduction to the logic of **dynamic programming**

Dynamic programming is an extremely powerful optimization technique that we apply in many lectures on this site

### 6.1.2 Outline of the Problem

The shortest path problem is one of finding how to traverse a **graph** from one specified node to another at minimum cost

Consider the following graph

![Graph Diagram](image)

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

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
6.1.3 Finding Least-Cost Paths

For large graphs we need a systematic solution.

Let $J(v)$ denote the minimum cost-to-go from node $v$, understood as the total cost from $v$ if we take the best route.

A, D, F, G at cost 8
Suppose that we know $J(v)$ for each node $v$, as shown below for the graph from the preceding example.

![Graph with labeled nodes and edges](image)

Note that $J(G) = 0$

The best path can now be found as follows

- Start at A
- From node $v$, move to any node that solves

$$
\min_{w \in F_v} \{c(v, w) + J(w)\}
$$

(6.1)

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 to find $J$?

Some thought will convince you that, for every node $v$, the function $J$ satisfies

$$
J(v) = \min_{w \in F_v} \{c(v, w) + J(w)\}
$$

(6.2)

This is known as the Bellman equation, after the mathematician Richard Bellman.
6.1.4 Solving for $J$

The standard algorithm for finding $J$ is to start with

$$J_0(v) = M \text{ if } v \neq \text{destination, else } J_0(v) = 0$$  (6.3)

where $M$ is some large number.

Now we use the following algorithm

1. Set $n = 0$
2. Set $J_{n+1}(v) = \min_{w \in E} \{c(v, w) + J_n(w)\}$ for all $v$
3. If $J_{n+1}$ and $J_n$ are not equal then increment $n$, go to 2

In general, this sequence converges to $J$ the proof is omitted.

6.1.5 Exercises

Exercise 1

Use the algorithm given above to find the optimal path (and its cost) for the following graph

You can put it in a Jupyter notebook cell and hit Shift-Enter it will be saved in the local directory as file graph.txt

```plaintext
% 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 4.49, 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 447.22, node39 136.32, node40 124.22
node23, node52 336.73, node26 2.66, node33 22.37
```
490 Chapter 6. Dynamic Programming

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, node32 4.22, node33 12.61
node31, node98 3350.98, node35 48.10, node43 246.24
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, node52 18.23, node46 0.28, node54 162.24
node45, node91 807.39, node46 1.56, node47 2.14
node46, node58 79.93, node47 3.68, node49 111.54
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, node64 102.41, node65 102.41, node66 123.03
node53, node72 22.85, node56 4.33, node67 88.35
node54, node88 967.59, node59 24.30, node63 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
Writing graph.txt

Here 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

and so on.

According to our calculations, the optimal path and its cost are like this:

Your code should replicate this result.

### 6.1.6 Solutions

#### Exercise 1

```python
def read_graph(in_file):
    """ Read in the graph from the data file. The graph is stored as a dictionary, where the keys are the nodes, and the values are a list of pairs (d, c), where d is a node and c is a number. If (d, c) is in the list for node n, then d can be reached from n at cost c. """ 
```

6.1. Shortest Paths
def update_J(J, graph):
    """The Bellman operator."
    next_J = {}
    for node in graph:
        if node == 'node99':
            next_J[node] = 0
        else:
            next_J[node] = min(cost + J[dest] for dest, cost in graph[node])
    return next_J

def print_best_path(J, graph):
    """Given a cost-to-go function, computes the best path. At each node n, the function prints the current location, looks at all nodes that can be reached from n, and moves to the node m which minimizes c + J[m], where c is the cost of moving to m."
    """
    sum_costs = 0
    current_location = 'node0'
    while current_location != 'node99':
        print(current_location)
        running_min = 1e100  # Any big number
        for destination, cost in graph[current_location]:
            cost_of_path = cost + J[destination]
            if cost_of_path < running_min:
                running_min = cost_of_path
                minimizer_cost = cost
                minimizer_dest = destination
        current_location = minimizer_dest
        sum_costs += minimizer_cost
        print('node99\n')
        print('Cost: ', sum_costs)

## Main loop

graph = read_graph('graph.txt')
M = 1e10
J = {}
for node in graph:
```python
J[node] = M
J['node99'] = 0

while True:
    next_J = update_J(J, graph)
    if next_J == J:
        break
    else:
        J = next_J

print_best_path(J, graph)
```

6.2 Job Search I: The McCall Search Model

Contents

- Job Search I: The McCall Search Model
  - Overview
  - The McCall Model
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.

### 6.2.1 Overview

The McCall search model \([McC70]\) 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 McCalls model and adopt the same solution method.

As well see, McCalls model is not only interesting in its own right but also an excellent vehicle for learning dynamic programming.

### 6.2.2 The McCall Model

An unemployed worker receives in each period a job offer at wage \(W_t\).

At time \(t\), our worker 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 wage sequence \(\{W_t\}\) is assumed to be iid with probability mass function \(p_1, \ldots, p_n\).

Here \(p_i\) is the probability of observing wage offer \(W_t = w_i\) in the set \(w_1, \ldots, w_n\).

The worker is infinitely lived and aims to maximize the expected discounted sum of earnings

\[
E \sum_{t=0}^{\infty} \beta^t Y_t
\]

The constant \(\beta\) lies in \((0, 1)\) and is called a **discount factor**.
The smaller is $\beta$, the more the worker discounts future utility relative to current utility.

The variable $Y_t$ is income, equal to:

- his wage $W_t$ when employed
- unemployment compensation $c$ when unemployed

**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.

**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 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 $V(w)$ be the total lifetime *value* accruing to an unemployed worker who enters the current period unemployed but with wage offer $w$ in hand.

More precisely, $V(w)$ denotes the value of the objective function (6.11) when an agent in this situation makes *optimal* decisions now and at all future points in time.

Of course $V(w)$ is not trivial to calculate because we don't yet know what decisions are optimal and what aren't!

But think of $V$ as a function that assigns to each possible wage $w$ the maximal lifetime value that can be obtained with that offer in hand.

A crucial observation is that this function $V$ must satisfy the recursion

$$V(w) = \max \left\{ \frac{w}{1-\beta}, c + \beta \sum_{i=1}^{n} V(w_i)p_i \right\}$$

(6.4)
for every possible \( w_i \) in \( w_1, \ldots, w_n \)

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 \( w \), since
  \[
  w + \beta w + \beta^2 w + \cdots = \frac{w}{1 - \beta}
  \]

- the second term inside the max operation is the **continuation value**, which is the lifetime payoff from rejecting the current offer and then behaving optimally in all subsequent periods.

If we optimize and pick the best of these two options, we obtain maximal lifetime value from today, given current offer \( w \).

But this is precisely \( V(w) \), which is the l.h.s. of \((6.4)\).

### The Optimal Policy

Suppose for now that we are able to solve \((6.4)\) 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 \((6.4)\).

The optimal action is best thought of as a **policy**, which is, in general, a map from states to actions.

In our case, the state is the current wage offer \( w \).

Given any \( w \), we can read off the corresponding best choice (accept or reject) by picking the max on the r.h.s. of \((6.4)\).

Thus, we have a map from \( \mathbb{R} \) to \( \{0, 1\} \), with 1 meaning accept and zero meaning reject.

We can write the policy as follows

\[
\sigma(w) := 1 \left\{ \frac{w}{1 - \beta} \geq c + \beta \sum_{i=1}^{n} V(w_i)p_i \right\}
\]

Here \( 1\{P\} = 1 \) if statement \( P \) is true and equals zero otherwise.

We can also write this as

\[
\sigma(w) := 1\{w \geq \bar{w}\}
\]

where

\[
\bar{w} := (1 - \beta) \left\{ c + \beta \sum_{i=1}^{n} V(w_i)p_i \right\}
\]

Here \( \bar{w} \) is a constant depending on \( \beta, c \) and the wage distribution, called the **reservation wage**.

The agent should accept if and only if the current wage offer exceeds the reservation wage.

Clearly, we can compute this reservation wage if we can compute the value function.
6.2.3 Computing the Optimal Policy: Take 1

To put the above ideas into action, we need to compute the value function at points $w_1, \ldots, w_n$.

In doing so, we can identify these values with the vector $v = (v_i)$ where $v_i := V(w_i)$.

In view of (6.4), this vector satisfies the nonlinear system of equations

$$v_i = \max \left\{ \frac{w_i}{1 - \beta}, c + \beta \sum_{i=1}^{n} v_i p_i \right\} \quad \text{for } i = 1, \ldots, n \quad (6.5)$$

It turns out that there is exactly one vector $v := (v_i)_{i=1}^{n}$ in $\mathbb{R}^n$ that satisfies this equation.

The Algorithm

To compute this vector, we proceed as follows:

Step 1: pick an arbitrary initial guess $v \in \mathbb{R}^n$

Step 2: compute a new vector $v' \in \mathbb{R}^n$ via

$$v'_i = \max \left\{ \frac{w_i}{1 - \beta}, c + \beta \sum_{i=1}^{n} v_i p_i \right\} \quad \text{for } i = 1, \ldots, n \quad (6.6)$$

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 $v$

This algorithm returns an arbitrarily good approximation to the true solution to (6.5), which represents the value function.

(Arbitrarily good means here that the approximation converges to the true solution as the tolerance goes to zero)

The Fixed Point Theory

What's the math behind these ideas?

First, one defines a mapping $T$ from $\mathbb{R}^n$ to itself via

$$Tv_i = \max \left\{ \frac{w_i}{1 - \beta}, c + \beta \sum_{i=1}^{n} v_i p_i \right\} \quad \text{for } i = 1, \ldots, n \quad (6.7)$$

(A new vector $Tv$ is obtained from given vector $v$ by evaluating the r.h.s. at each $i$)
One can show that the conditions of the Banach contraction mapping theorem are satisfied by $T$ as a self-mapping on $\mathbb{R}^n$

One implication is that $T$ has a unique fixed point in $\mathbb{R}^n$

Moreover, it is immediate from the definition of $T$ that this fixed point is precisely the value function

The iterative algorithm presented above corresponds to iterating with $T$ from some initial guess $v$

The Banach contraction mapping theorem tells us that this iterative process generates a sequence that converges to the fixed point

**Implementation**

Let's start with some imports

```python
import numpy as np
from numba import jit
import matplotlib.pyplot as plt
import quantecon as qe
from quantecon.distributions import BetaBinomial
```

Here's the distribution of wage offers well work with

```python
n, a, b = 50, 200, 100
w_min, w_max = 10, 60
w_vals = np.linspace(w_min, w_max, n+1)
dist = BetaBinomial(n, a, b)
p_vals = dist.pdf()

fig, ax = plt.subplots(figsize=(9, 6.5))
ax.stem(w_vals, p_vals, label='$p_i$')
ax.set_xlabel('wages')
ax.set_ylabel('probabilities')

plt.show()
```
First let's have a look at the sequence of approximate value functions that the algorithm above generates.

Default parameter values are embedded in the function.

Our initial guess $v$ is the value of accepting at every given wage.

```python
def plot_value_function_seq(ax,
c=25,
$\beta=0.99$,
w_vals=w_vals,
p_vals=p_vals,
num_plots=6):
    v = w_vals / (1 - $\beta$)
    v_next = np.empty_like(v)
    for i in range(num_plots):
        ax.plot(w_vals, v, label=f"iterate {i}")
        # Update guess
        for j, w in enumerate(w_vals):
            stop_val = w / (1 - $\beta$)
            cont_val = c + $\beta$ * np.sum(v * p_vals)
            v_next[j] = max(stop_val, cont_val)
        v[:] = v_next
    ax.legend(loc='lower right')
```

6.2. Job Search I: The McCall Search Model 499
Here's more serious iteration effort, that continues until measured deviation between successive iterates is below $\text{tol}$

Well be using JIT compilation via Numba to turbo charge our loops

```python
@jit(nopython=True)
def compute_reservation_wage(c=25,
                           $\beta=0.99$,
                           w_vals=w_vals,
                           p_vals=p_vals,
                           max_iter=500,
                           tol=1e-6):

    # == First compute the value function ==#

    v = w_vals / (1 - $\beta$)
    v_next = np.empty_like(v)
    i = 0
```
error = tol + 1
while i < max_iter and error > tol:
    for j, w in enumerate(w_vals):
        stop_val = w / (1 - β)
        cont_val = c + β * np.sum(v * p_vals)
        v_next[j] = max(stop_val, cont_val)
    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 * p_vals))

Let's compute the reservation wage at the default parameters

```python
compute_reservation_wage()
```

```
47.316499710024964
```

**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 $\beta$ and $c$.

```python
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):
        R[i, j] = compute_reservation_wage(c=c, β=β)
```

```python
fig, ax = plt.subplots(figsize=(10, 5.7))
cs1 = ax.contourf(c_vals, β_vals, R.T, alpha=0.75)
ctrl1 = ax.contour(c_vals, β_vals, R.T)
plt.clabel(ctrl1, inline=1, fontsize=13)
plt.colorbar(cs1, ax=ax)

ax.set_title("reservation wage")
ax.set_xlabel("$c$", fontsize=16)
ax.set_ylabel("$\beta$", fontsize=16)
```

6.2. Job Search I: The McCall Search Model
As expected, the reservation wage increases both with patience and with unemployment compensation.

**6.2.4 Computing the Optimal Policy: Take 2**

The approach to dynamic programming just described is very standard and broadly applicable.

For this particular problem, there is also an easier way, which circumvents the need to compute the value function.

Let $\psi$ denote the value of not accepting a job in this period but then behaving optimally in all subsequent periods.

That is,

$$\psi = c + \beta \sum_{i=1}^{n} V(w_i)p_i$$  \hspace{1cm} (6.8)

where $V$ is the value function.
By the Bellman equation, we then have

\[ V(w_i) = \max \left\{ \frac{w_i}{1 - \beta}, \psi \right\} \]

Substituting this last equation into (6.8) gives

\[ \psi = c + \beta \sum_{i=1}^{n} \max \left\{ \frac{w_i}{1 - \beta}, \psi \right\} p_i \]  

(6.9)

This is a nonlinear equation that we can solve for \( \psi \)

The natural solution method for this kind of nonlinear equation is iterative

That is,

Step 1: pick an initial guess \( \psi \)

Step 2: compute the update \( \psi' \) via

\[ \psi' = c + \beta \sum_{i=1}^{n} \max \left\{ \frac{w_i}{1 - \beta}, \psi \right\} p_i \]  

(6.10)

Step 3: calculate the deviation \( |\psi - \psi'| \)

Step 4: if the deviation is larger than some fixed tolerance, set \( \psi = \psi' \) and go to step 2, else continue

Step 5: return \( \psi \)

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 are iterating on a single number, rather than an \( n \)-vector

Here is an implementation:

```python
@jit(nopython=True)
def compute_reservation_wage_two(c=25, 
    beta=0.99, 
    w_vals=w_vals, 
    p_vals=p_vals, 
    max_iter=500, 
    tol=1e-5):

    # == First compute \( \psi \) == #
    psi = np.sum(w_vals * p_vals) / (1 - beta)
    i = 0
    error = tol + 1
    while i < max_iter and error > tol:
        s = np.maximum(w_vals / (1 - beta), psi)
        psi_next = c + beta * np.sum(s * p_vals)
error = np.abs(ψ_next - ψ)
i += 1
ψ = ψ_next

# == Now compute the reservation wage == #
return (1 - β) * (c + β * ψ)

You can use this code to solve the exercise below

6.2.5 Exercises

Exercise 1

Compute the average duration of unemployment when $\beta = 0.99$ and $c$ takes the following values

$$c_vals = \text{np.linspace}(10, 40, 25)$$

That is, start the agent off as unemployed, computed 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 $c_vals$

6.2.6 Solutions

Exercise 1

Here's one solution

```python
cdf = np.cumsum(p_vals)

@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_vals[qe.random.draw(cdf)]
        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):
```

Chapter 6. Dynamic Programming
```python
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):
    w_bar = compute_reservation_wage_two(c=c)
    stop_times[i] = compute_mean_stopping_time(w_bar)

fig, ax = plt.subplots(figsize=(9, 6.5))
ax.plot(c_vals, stop_times, label="mean unemployment duration")
ax.set(xlabel="unemployment compensation", ylabel="months")
ax.legend()
plt.show()
```

6.2. Job Search I: The McCall Search Model
6.3 Job Search II: Search and Separation

Contents

- Job Search II: Search and Separation
  - Overview
  - The Model
  - Solving the Model using Dynamic Programming
  - Implementation
  - The Reservation Wage
  - Exercises
  - Solutions

6.3.1 Overview

Previously we looked at the McCall job search model [McC70] as a way of understanding unemployment and worker decisions.

One unrealistic feature of the model is that every job is permanent.

In this lecture we extend the McCall model by introducing job separation.

Once separation enters the picture, the agent comes to view

• the loss of a job as a capital loss, and
• a spell of unemployment as an investment in searching for an acceptable job.

6.3.2 The Model

The model 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

$$
E \sum_{t=0}^{\infty} \beta^t u(Y_t)
$$

(6.11)
The only difference from the *baseline model* is that we've added some flexibility over preferences by introducing a utility function $u$

It satisfies $u' > 0$ and $u'' < 0$

**Timing and Decisions**

Here's what happens at the start of a given period in our model with search and separation.

If currently employed, the worker consumes his wage $w$, receiving utility $u(w)$

If currently unemployed, he

- receives and consumes unemployment compensation $c$
- receives an offer to start work next period at a wage $w'$ drawn from a known distribution $p_1, \ldots, p_n$

He can either accept or reject the offer.

If he accepts the offer, he enters next period employed with wage $w'$

If he rejects the offer, he enters next period unemployed.

When employed, the agent faces a constant probability $\alpha$ of becoming unemployed at the end of the period.

(Note: we do not allow for job search while employed; this topic is taken up in a later lecture.)

### 6.3.3 Solving the Model using Dynamic Programming

Let

- $V(w)$ be the total lifetime value accruing to a worker who enters the current period employed with wage $w$
- $U$ be the total lifetime value accruing to a worker who is unemployed this period.

Here *value* means the value of the objective function (6.11) when the worker makes optimal decisions at all future points in time.

Suppose for now that the worker can calculate the function $V$ and the constant $U$ and use them in his decision making.

Then $V$ and $U$ should satisfy

$$V(w) = u(w) + \beta[(1 - \alpha)V(w) + \alpha U]$$

and

$$U = u(c) + \beta \sum \max \{U, V(w_i)\} p_i$$

Let's interpret these two equations in light of the fact that today's tomorrow is tomorrow's today.
• The left hand sides of equations (6.12) and (6.13) are the values of a worker in a particular situation today.
• The right hand sides of the equations are the discounted (by $\beta$) expected values of the possible situations that worker can be in tomorrow.
• But tomorrow the worker can be in only one of the situations whose values today are on the left sides of our two equations.

Equation (6.13) incorporates the fact that a currently unemployed worker will maximize his own welfare. In particular, if his next period wage offer is $w'$, he will choose to remain unemployed unless $U < V(w')$.

Equations (6.12) and (6.13) are the Bellman equations for this model.

Equations (6.12) and (6.13) provide enough information to solve out for both $V$ and $U$.

Before discussing this, however, lets make a small extension to the model.

### Stochastic Offers

Lets suppose now that unemployed workers dont always receive job offers.

Instead, lets suppose that unemployed workers only receive an offer with probability $\gamma$.

If our worker does receive an offer, the wage offer is drawn from $p$ as before.

He either accepts or rejects the offer.

Otherwise the model is the same.

With some thought, you will be able to convince yourself that $V$ and $U$ should now satisfy

$$ V(w) = u(w) + \beta[(1 - \alpha)V(w) + \alpha U] \quad (6.14) $$

and

$$ U = u(c) + \beta(1 - \gamma)U + \beta\gamma \sum_i \max\{U, V(w_i)\} p_i \quad (6.15) $$

### Solving the Bellman Equations

Well 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 $U$ and $V$
2. plug these guesses into the right hand sides of (6.14) and (6.15)
3. update the left hand sides from this rule and then repeat
In other words, we are iterating using the rules

\[ V_{n+1}(w_i) = u(w_i) + \beta [(1 - \alpha)V_n(w_i) + \alpha U_n] \]  

(6.16)

and

\[ U_{n+1} = u(c) + \beta (1 - \gamma)U_n + \beta \gamma \sum_i \max\{U_n, V_n(w_i)\}p_i \]  

(6.17)

starting from some initial conditions \( U_0, V_0 \)

As before, the system always converges to the true solutions in this case, the \( V \) and \( U \) that solve (6.14) and (6.15)

A proof can be obtained via the Banach contraction mapping theorem

### 6.3.4 Implementation

Let's implement this iterative process

In the code you'll see that we use a class to store the various parameters and other objects associated with a given model

This helps to tidy up the code and provides an object that's easy to pass to functions

The default utility function is a CRRA utility function

In places we use just in time compilation via Numba to achieve good performance

```python
import numpy as np
from quantecon.distributions import BetaBinomial
from numba import jit

# 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
                 ...)
```

6.3. Job Search II: Search and Separation
\[
\begin{align*}
\gamma &= 0.7, \quad \text{# Job offer rate} \\
c &= 6.0, \quad \text{# Unemployment compensation} \\
\sigma &= 2.0, \quad \text{# Utility parameter} \\
w_{\text{vec}} &= \text{None}, \quad \text{# Possible wage values} \\
p_{\text{vec}} &= \text{None} \quad \text{# Probabilities over w_{\text{vec}}} \\
\end{align*}
\]

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  
    self.w_vec = np.linspace(10, 20, n)  # wages between 10 and 20  
    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 + \n            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
The approach is to iterate until successive iterates are closer together than some small tolerance level. We then return the current iterate as an approximate solution.

Let's plot the approximate solutions $U$ and $V$ to see what they look like. We'll use the default parameterizations found in the code above.

```python
import matplotlib.pyplot as plt

mcm = McCallModel()
V, U = solve_mccall_model(mcm)

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(mcm.w_vec, V, 'b-', lw=2, alpha=0.7, label='$V$')
ax.plot(mcm.w_vec, [U]*len(mcm.w_vec), 'g-', lw=2, alpha=0.7, label='$U$')
ax.set_xlim(min(mcm.w_vec), max(mcm.w_vec))
ax.legend(loc='upper left')
ax.grid()

plt.show()
```

Here's the plot this code produces.
The value $V$ is increasing because higher $w$ generates a higher wage flow conditional on staying employed.

### 6.3.5 The Reservation Wage

Once $V$ and $U$ are known, the agent can use them to make decisions in the face of a given wage offer.

- If $V(w) > U$, then working at wage $w$ is preferred to unemployment.
- If $V(w) < U$, then remaining unemployed will generate greater lifetime value.

Suppose in particular that $V$ crosses $U$ (as it does in the preceding figure).

Then, since $V$ is increasing, there is a unique smallest $w$ in the set of possible wages such that $V(w) \geq U$.

We denote this wage $\bar{w}$ and call it the reservation wage.

Optimal behavior for the worker is characterized by $\bar{w}$:

- if the wage offer $w$ in hand is greater than or equal to $\bar{w}$, then the worker accepts.
- if the wage offer $w$ in hand is less than $\bar{w}$, then the worker rejects.

Here's a function called `compute_reservation_wage` that takes an instance of a McCall model and returns the reservation wage associated with a given model.

It uses `np.searchsorted` to obtain the first $w$ in the set of possible wages such that $V(w) > U$.

If $V(w) < U$ for all $w$, then the function returns `np.inf`.
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

Lets use it to look at how the reservation wage varies with parameters

In each instance below well show you a figure and then ask you to reproduce it in the exercises

**The Reservation Wage and Unemployment Compensation**

First, lets 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.

**The Reservation Wage and Discounting**

Next, let's investigate how $\bar{w}$ varies with the discount rate.

The next figure plots the reservation wage associated with different values of $\beta$. 
Again, the results are intuitive: More patient workers will hold out for higher wages

**The Reservation Wage and Job Destruction**

Finally, let's look at how \( \hat{w} \) varies with the job separation rate \( \alpha \).

Higher \( \alpha \) 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

6.3.6 Exercises

Exercise 1
Reproduce all the reservation wage figures shown above

Exercise 2
Plot the reservation wage against the job offer rate $\gamma$
Use

```
grid_size = 25
\gamma_vals = np.linspace(0.05, 0.95, grid_size)
```
Interpret your results
6.3.7 Solutions

Exercise 1

Using the `compute_reservation_wage` function mentioned earlier in the lecture, we can create an array for reservation wages for different values of $c$, $\beta$ and $\alpha$ and plot the results like so

```python
grid_size = 25
c_vals = np.linspace(2, 12, grid_size)  # values of unemployment compensation
w_bar_vals = np.empty_like(c_vals)
mcm = McCallModel()
fig, ax = plt.subplots(figsize=(10, 6))

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')
ax.set_ylabel('reservation wage')
txt = r'$\bar{w}$ as a function of $c$'
ax.plot(c_vals, w_bar_vals, 'b-', lw=2, alpha=0.7, label=txt)
ax.legend(loc='upper left')
ax.grid()

plt.show()
```

Exercise 2

Similar to above, we can plot $\bar{w}$ against $\gamma$ as follows

```python
import matplotlib.pyplot as plt

grid_size = 25
_vals = np.linspace(0.05, 0.95, grid_size)
_w_bar_vals = np.empty_like(_vals)
mcm = McCallModel()
fig, ax = plt.subplots(figsize=(10, 6))

for i, _ in enumerate(_vals):
    mcm._ = _
    _w_bar = compute_reservation_wage(mcm)
    _w_bar_vals[i] = _w_bar

ax.set_xlabel('job offer rate')
ax.set_ylabel('reservation wage')
ax.set_xlim(_vals.min(), _vals.max())
```
As expected, the reservation wage increases in $\gamma$

This is because higher $\gamma$ translates to a more favorable job search environment

Hence workers are less willing to accept lower offers

### 6.4 A Problem that Stumped Milton Friedman

(and that Abraham Wald solved by inventing sequential analysis)
A dynamic programming approach

Implementation

Analysis

Comparison with Neyman-Pearson formulation

Co-authors: Chase Coleman

6.4.1 Overview

This lecture describes a statistical decision problem encountered by Milton Friedman and W. Allen Wal- lis during World War II when they were analysts at the U.S. Governments Statistical Research Group at Columbia University

This problem led Abraham Wald [Wal47] to formulate sequential analysis, an approach to statistical deci- sion problems intimately related to dynamic programming

In this lecture, we apply dynamic programming algorithms to Friedman and Wallis and Walds 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 Walds sequential probability ratio test
- The power of a statistical test
- The critical region of a statistical test
- A uniformly most powerful test

6.4.2 Origin of the problem

On pages 137-139 of his 1998 book Two Lucky People with Rose Friedman [FF98], Milton Friedman described a problem presented to him and Allen Wallis during World War II, when they worked at the US Governments Statistical Research Group at Columbia University

Lets 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 inferio to that of B and conversely 0 or 1 to B. The Navy asks the statistician how to conduct the test and how to analyze the results.

The standard statistical answer was to specify a number of firings (say 1,000) and a pair of percentages (e.g., 53% and 47%) and tell the client that if A receives a 1 in more than 53% of the firings, it can be regarded as superior; if it receives a 1 in fewer than 47%, B can be regarded as superior; if the percentage is between 47% and 53%, neither can be so regarded.

When Allen Wallis was discussing such a problem with (Navy) Captain Garret L. Schyler, the captain objected that such a test, to quote from Allen’s account, may prove wasteful. If a wise and seasoned ordnance officer like Schyler were on the premises, he would see after the first few thousand or even few hundred [rounds] that the experiment need not be completed either because the new method is obviously inferior or because it is obviously superior beyond what was hoped for . . .

Friedman and Wallis struggled with the problem but, after realizing that they were not able to solve it, described the problem to Abraham Wald

That started Wald on the path that led him to *Sequential Analysis* [*Wal47*]

Well formulate the problem using dynamic programming

### 6.4.3 A dynamic programming approach

The following presentation of the problem closely follows Dmitri Berskekass treatment in *Dynamic Programming and Stochastic Control* [*Ber75*]

A decision maker observes iid draws of a random variable $z$

He (or she) wants to know which of two probability distributions $f_0$ or $f_1$ governs $z$

After a number of draws, also to be determined, he makes a decision as to which of the distributions is generating the draws he observers

To help formalize the problem, let $x \in \{x_0, x_1\}$ be a hidden state that indexes the two distributions:

$$
\mathbb{P}\{z = v \mid x\} = \begin{cases} 
  f_0(v) & \text{if } x = x_0, \\
  f_1(v) & \text{if } x = x_1
\end{cases}
$$

Before observing any outcomes, the decision maker believes that the probability that $x = x_0$ is

$$
p_{-1} = \mathbb{P}\{x = x_0 \mid \text{no observations}\} \in (0, 1)
$$

After observing $k + 1$ observations $z_k, z_{k-1}, \ldots, z_0$, he updates this value to

$$
p_k = \mathbb{P}\{x = x_0 \mid z_k, z_{k-1}, \ldots, z_0\},
$$

which is calculated recursively by applying Bayes law:

$$
p_{k+1} = \frac{p_k f_0(z_{k+1})}{p_k f_0(z_{k+1}) + (1 - p_k) f_1(z_{k+1})}, \quad k = -1, 0, 1, \ldots
$$

After observing $z_k, z_{k-1}, \ldots, z_0$, the decision maker believes that $z_{k+1}$ has probability distribution

$$
f(v) = p_k f_0(v) + (1 - p_k) f_1(v)
$$
This is a mixture of distributions \( f_0 \) and \( f_1 \), with the weight on \( f_0 \) being the posterior probability that \( x = x_0 \).\footnote{Because the decision maker believes that \( z_{k+1} \) is drawn from a mixture of two i.i.d. distributions, he does \textit{not} believe that the sequence \([z_{k+1}, z_{k+2}, \ldots]\) is i.i.d. Instead, he believes that it is \textit{exchangeable}. See [Kre88] chapter 11, for a discussion of exchangeability.}

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

\[
f(z; a, b) = \frac{\Gamma(a + b)z^{a-1}(1-z)^{b-1}}{\Gamma(a)\Gamma(b)} \quad \text{where} \quad \Gamma(t) := \int_0^\infty x^{t-1}e^{-x}dx
\]

We'll discretize this distribution to make it more straightforward to work with.

The next figure shows two discretized beta distributions in the top panel.

The bottom panel presents mixtures of these distributions, with various mixing probabilities \( p_k \).

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as st

def make_distribution_plots(f0, f1):
    """This generates the figure that shows the initial versions of the distributions and plots their combinations."
    fig, axes = plt.subplots(2, figsize=(10, 8))

    axes[0].set_title("Original Distributions")
    axes[0].plot(f0, lw=2, label="$f_0$")
    axes[0].plot(f1, lw=2, label="$f_1$")

    axes[1].set_title("Mixtures")
    for p in 0.25, 0.5, 0.75:
        y = p * f0 + (1 - p) * f1
        axes[1].plot(y, lw=2, label=f"$p_k = (p)$")

    for ax in axes:
        ax.legend(fontsize=14)
        ax.set_xlabel("$k$ values", fontsize=14)
        ax.set_ylabel("probability of $z_k$", fontsize=14)
        ax.set_ylim(0, 0.07)

    plt.tight_layout()
    plt.show()

p_m1 = np.linspace(0, 1, 50)
f0 = np.clip(st.beta.pdf(p_m1, a=1, b=1), 1e-8, np.inf)
f0 = f0 / np.sum(f0)
f1 = np.clip(st.beta.pdf(p_m1, a=9, b=9), 1e-8, np.inf)
```

6.4. A Problem that Stumped Milton Friedman
\[
f_1 = f_1 / np.sum(f_1)
\]

\[
\text{make_distribution_plots}(f_0, f_1)
\]

Losses and costs

After observing \( z_k, z_{k-1}, \ldots, z_0 \), the decision maker chooses among three distinct actions:

- He decides that \( x = x_0 \) and draws no more zs
- He decides that \( x = x_1 \) and draws no more zs
- 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 \( x = x_0 \) when actually \( x = x_1 \)
- A loss \( L_1 \) if he decides \( x = x_1 \) when actually \( x = x_0 \)
- A cost \( c \) if he postpones deciding and chooses instead to draw another \( z \)
Digression on type I and type II errors

If we regard $x = x_0$ as a null hypothesis and $x = x_1$ 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 $x = x_0$ 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

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 $p$ is close to 1

Then our prior beliefs and the evidence so far point strongly to $x = x_0$

If, on the other hand, $p$ is close to 0, then $x = x_1$ is strongly favored.

Finally, if $p$ 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:

```
accept x_1

\beta

draw again

\alpha

accept x_0

0

values of p

1
```

As well see, this is indeed the correct form of the decision rule.

The key problem is to determine the threshold values $\alpha, \beta$, 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 $\alpha$ or $\beta$.

A Bellman equation

Let $J(p)$ be the total loss for a decision maker with current belief $p$ who chooses optimally.

With some thought, you will agree that $J$ should satisfy the Bellman equation:

$$J(p) = \min \left\{ (1 - p)L_0, \ pL_1, \ c + \mathbb{E}[J(p')] \right\}$$

(6.18)
where $p'$ is the random variable defined by

$$p' = \frac{pf_0(z)}{pf_0(z) + (1-p)f_1(z)}$$

when $p$ is fixed and $z$ is drawn from the current best guess, which is the distribution $f$ defined by

$$f(v) = pf_0(v) + (1-p)f_1(v)$$

In the Bellman equation, minimization is over three actions:

1. accept $x_0$
2. accept $x_1$
3. postpone deciding and draw again

Let

$$A(p) := \mathbb{E}[J(p')]$$

Then we can represent the Bellman equation as

$$J(p) = \min \{ (1-p)L_0, pL_1, c + A(p) \}$$

where $p \in [0, 1]$

Here

- $(1-p)L_0$ is the expected loss associated with accepting $x_0$ (i.e., the cost of making a type II error)
- $pL_1$ is the expected loss associated with accepting $x_1$ (i.e., the cost of making a type I error)
- $c + A(p)$ is 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

$$(1-p)L_0 < \min \{ pL_1, c + A(p) \} \text{ if } p \geq \alpha$$

and

$$pL_1 < \min \{ (1-p)L_0, c + A(p) \} \text{ if } p \leq \beta$$

The optimal decision rule is then

- accept $x = x_0$ if $p \geq \alpha$
- accept $x = x_1$ if $p \leq \beta$
- draw another $z$ if $\beta \leq p \leq \alpha$

Our aim is to compute the value function $J$, and from it the associated cutoffs $\alpha$ and $\beta$

One sensible approach is to write the three components of $J$ that appear on the right side of the Bellman equation as separate functions

Later, doing this will help us obey the **don't repeat yourself (DRY)** golden rule of coding
6.4.4 Implementation

Let's code this problem up and solve it.

To approximate the value function that solves Bellman equation (6.18), we use value function iteration.

- For earlier examples of this technique see the shortest path, job search or optimal growth lectures.

As in the optimal growth lecture, to approximate a continuous value function:

- We iterate at a finite grid of possible values of \( p \).
- When we evaluate \( A(p) \) between grid points, we use linear interpolation.

This means that to evaluate \( J(p) \) where \( p \) is not a grid point, we must use two points:

- First, we use the largest of all the grid points smaller than \( p \), and call it \( p_i \).
- Second, we use the grid point immediately after \( p \), named \( p_{i+1} \), to approximate the function value as

\[
J(p) = J(p_i) + (p - p_i) \frac{J(p_{i+1}) - J(p_i)}{p_{i+1} - p_i}
\]

In one dimension, you can think of this as simply drawing a line between each pair of points on the grid.

Here's the code:

```python
import scipy.interpolate as interp
import quantecon as qe

def expect_loss_choose_0(p, L0):
    """For a given probability return expected loss of choosing model 0"
    return (1 - p) * L0

def expect_loss_choose_1(p, L1):
    """For a given probability return expected loss of choosing model 1"
    return p * L1

def EJ(p, f0, f1, J):
    """
    Evaluates the expectation of our value function J. To do this, we need the current probability 
    that model 0 is correct (p), the distributions (f0, f1), and the function J.
    """
    # Get the current distribution we believe (p*f0 + (1-p)*f1)
    curr_dist = p * f0 + (1 - p) * f1

    # Get tomorrow's expected distribution through Bayes law
    tpl_dist = np.clip((p * f0) / (p * f0 + (1 - p) * f1), 0, 1)

    # Evaluate the expectation
    EJ = curr_dist @ J(tpl_dist)
    return EJ

def expect_loss_cont(p, c, f0, f1, J):
    return c + EJ(p, f0, f1, J)
```

6.4. A Problem that Stumped Milton Friedman
def bellman_operator(pgrid, c, f0, f1, L0, L1, J):
    
    Evaluates the value function for a given continuation value function; that is, evaluates
    
    \[ J(p) = \min((1 - p) L_0, p L_1, c + E J(p')) \]
    
    Uses linear interpolation between points.
    
    m = np.size(pgrid)
    assert m == np.size(J)

    J_out = np.zeros(m)
    J_interp = interp.UnivariateSpline(pgrid, J, k=1, ext=0)

    for (p_ind, p) in enumerate(pgrid):
        # Payoff of choosing model 0
        p_c_0 = expect_loss_choose_0(p, L0)
        p_c_1 = expect_loss_choose_1(p, L1)
        p_con = expect_loss_cont(p, c, f0, f1, J_interp)

        J_out[p_ind] = min(p_c_0, p_c_1, p_con)

    return J_out

# == Now run at given parameters == #

# First set up distributions
p_m1 = np.linspace(0, 1, 50)
f0 = np.clip(st.beta.pdf(p_m1, a=1, b=1), 1e-8, np.inf)
f0 = f0 / np.sum(f0)
f1 = np.clip(st.beta.pdf(p_m1, a=9, b=9), 1e-8, np.inf)
f1 = f1 / np.sum(f1)

# Build a grid
pg = np.linspace(0, 1, 251)

# Turn the Bellman operator into a function with one argument
bell_op = lambda vf: bellman_operator(pg, 0.5, f0, f1, 5.0, 5.0, vf)

# Pass it to qe’s built in iteration routine
J = qe.compute_fixed_point(bell_op,
                            np.zeros(pg.size), # Initial guess
                            error_tol=1e-6,
                            verbose=True,
                            print_skip=5)

Running it produces the following output on our machine

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Distance</th>
<th>Elapsed (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The distance column shows the maximal distance between successive iterates. This converges to zero quickly, indicating a successful iterative procedure. Iteration terminates when the distance falls below some threshold.

A more sophisticated implementation

Now for some gentle criticisms of the preceding code. By writing the code in terms of functions, we have to pass around some things that are constant throughout the problem:

- \( c, f_0, f_1, L_0, \) and \( L_1 \)

So now let's turn our simple script into a class. This will allow us to simplify the function calls and make the code more reusable.

We shall construct a class that:

- stores all of our parameters for us internally
- incorporates many of the same functions that we used above
- allows us, in addition, to simulate draws and the decision process under different prior beliefs

```python
class WaldFriedman:
    """
    Insert relevant docstrings here
    """
    def __init__(self, c, L0, L1, f0, f1, m=25):
        self.c = c
        self.L0, self.L1 = L0, L1
        self.m = m
        self.pgrid = np.linspace(0.0, 1.0, m)
        # Renormalize distributions so nothing is "too" small
        f0 = np.clip(f0, 1e-8, 1-1e-8)
        f1 = np.clip(f1, 1e-8, 1-1e-8)
        self.f0 = f0 / np.sum(f0)
        self.f1 = f1 / np.sum(f1)
        self.J = np.zeros(m)

    def current_distribution(self, p):
        """
        This function takes a value for the probability with which the correct model is model 0 and returns the mixed distribution that corresponds with that belief.
        """
        return p*self.f0 + (1-p)*self.f1
```

6.4. A Problem that Stumped Milton Friedman
def bayes_update_k(self, p, k):
    """
    This function takes a value for p, and a realization of the
    random variable and calculates the value for p tomorrow.
    """
    f0_k = self.f0[k]
    f1_k = self.f1[k]
    p_tp1 = p * f0_k / (p * f0_k + (1 - p) * f1_k)
    return np.clip(p_tp1, 0, 1)

def bayes_update_all(self, p):
    """
    This is similar to 'bayes_update_k' except it returns a
    new value for p for each realization of the random variable
    """
    return np.clip(p * self.f0 / (p * self.f0 + (1 - p) * self.f1), 0, 1)

def payoff_choose_f0(self, p):
    """For a given probability specify the cost of accepting model 0"
    return (1 - p) * self.L0

def payoff_choose_f1(self, p):
    """For a given probability specify the cost of accepting model 1"
    return p * self.L1

def EJ(self, p, J):
    """
    This function evaluates the expectation of the value function
    at period t+1. It does so by taking the current probability
    distribution over outcomes:

    \[ p(z_{k+1}) = p_k f_0(z_{k+1}) + (1-p_k) f_1(z_{k+1}) \]

    and evaluating the value function at the possible states
    tomorrow \( J(p_{(t+1)}) \) where

    \[ p_{(t+1)} = p f0 / ( p f0 + (1-p) f1) \]

    Parameters
    ---------
    p : Scalar(Float64)
    The current believed probability that model 0 is the true
    model.
    J : Function
    The current value function for a decision to continue

    Returns
    ------
    EJ : Scalar(Float64)
    The expected value of the value function tomorrow
"""
# Pull out information
f0, f1 = self.f0, self.f1

# Get the current believed distribution and tomorrows possible dists
# Need to clip to make sure things don't blow up (go to infinity)
curr_dist = self.current_distribution(p)
tpl1_dist = self.bayes_update_all(p)

# Evaluate the expectation
EJ = curr_dist @ J(tpl1_dist)
return EJ

def payoff_continue(self, p, J):
    """
    For a given probability distribution and value function give
cost of continuing the search for correct model
    """
    return self.c + self.EJ(p, J)

def bellman_operator(self, J):
    """
    Evaluates the value function for a given continuation value
    function; that is, evaluates

    \[ J(p) = \min(pL0, (1-p)L1, c + E[J(p')] \] 

    Uses linear interpolation between points
    """
    payoff_choose_f0 = self.payoff_choose_f0
    payoff_choose_f1 = self.payoff_choose_f1
    payoff_continue = self.payoff_continue
    c, L0, L1, f0, f1 = self.c, self.L0, self.L1, self.f0, self.f1
    m, pgrid = self.m, self.pgrid
    J_out = np.empty(m)
    J_interp = interp.UnivariateSpline(pgrid, J, k=1, ext=0)

    for (p_ind, p) in enumerate(pgrid):
        # Payoff of choosing model 0
        p_c_0 = payoff_choose_f0(p)
        p_c_1 = payoff_choose_f1(p)
        p_con = payoff_continue(p, J_interp)

        J_out[p_ind] = min(p_c_0, p_c_1, p_con)

    return J_out

def solve_model(self):
    J = qe.compute_fixed_point(self.bellman_operator, np.zeros(self.m),
        error_tol=1e-7, verbose=False)

6.4. A Problem that Stumped Milton Friedman
def find_cutoff_rule(self, J):
    
    This function takes a value function and returns the corresponding
cutoffs of where you transition between continue and choosing a
specific model
    
    payoff_choose_f0 = self.payoff_choose_f0
    payoff_choose_f1 = self.payoff_choose_f1
    m, pgrid = self.m, self.pgrid

    # Evaluate cost at all points on grid for choosing a model
    p_c_0 = payoff_choose_f0(pgrid)
    p_c_1 = payoff_choose_f1(pgrid)

    # 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)
    lb = pgrid[np.searchsorted(p_c_1 - J, 1e-10) - 1]
    ub = pgrid[np.searchsorted(J - p_c_0, -1e-10)]

    return (lb, ub)

def simulate(self, f, p0=0.5):
    
    This function takes an initial condition and simulates until it
stops (when a decision is made).
    
    # Check whether vf is computed
    if np.sum(self.J) < 1e-8:
        self.solve_model()

    # Unpack useful info
    lb, ub = self.find_cutoff_rule(self.J)
    update_p = self.bayes_update_k
    curr_dist = self.current_distribution

    # Initialize a couple useful variables
    decision_made = False
    p = p0
    t = 0

    while decision_made is False:
        # Maybe should specify which distribution is correct one so that
        # the draws come from the "right" distribution
        k = int(qe.random.draw(np.cumsum(f)))
        t = t+1
        p = update_p(p, k)
        if p < lb:
            decision_made = True
            decision = 1
        elif p > ub:
            decision_made = False
            decision = 0
        elif p_c_0 <= p < p_c_1:
            decision_made = True
            decision = 2
decision_made = True
decision = 0

return decision, p, t

def simulate_tdgp_f0(self, p0=0.5):
    """
    Uses the distribution f0 as the true data generating process
    """
    decision, p, t = self.simulate(self.f0, p0)

    if decision == 0:
        correct = True
    else:
        correct = False

    return correct, p, t

def simulate_tdgp_f1(self, p0=0.5):
    """
    Uses the distribution f1 as the true data generating process
    """
    decision, p, t = self.simulate(self.f1, p0)

    if decision == 1:
        correct = True
    else:
        correct = False

    return correct, p, t

def stopping_dist(self, ndraws=250, tdgp="f0"):
    """
    Simulates repeatedly to get distributions of time needed to make a decision and how often they are correct.
    """
    if tdgp == "f0":
        simfunc = self.simulate_tdgp_f0
    else:
        simfunc = self.simulate_tdgp_f1

    # Allocate space
    tdist = np.empty(ndraws, int)
    cdist = np.empty(ndraws, bool)

    for i in range(ndraws):
        correct, p, t = simfunc()
        tdist[i] = t
        cdist[i] = correct

    return cdist, tdist

6.4. A Problem that Stumped Milton Friedman  531
Now let's use our class to solve Bellman equation (6.18) and verify that it gives similar output

```python
# Set up distributions
p_m1 = np.linspace(0, 1, 50)
f0 = np.clip(st.beta.pdf(p_m1, a=1, b=1), 1e-8, np.inf)
f0 = f0 / np.sum(f0)
f1 = np.clip(st.beta.pdf(p_m1, a=9, b=9), 1e-8, np.inf)
f1 = f1 / np.sum(f1)

# Create an instance
wf = WaldFriedman(0.5, 5.0, 5.0, f0, f1, m=251)

# Compute the value function
wfJ = qe.compute_fixed_point(wf.bellman_operator, np.zeros(251),
                            error_tol=1e-6, verbose=True, print_skip=5)
```

We get the same output in terms of distance

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Distance</th>
<th>Elapsed (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8.042e-02</td>
<td>4.754e-02</td>
</tr>
<tr>
<td>10</td>
<td>6.418e-04</td>
<td>8.886e-02</td>
</tr>
<tr>
<td>15</td>
<td>4.482e-06</td>
<td>1.305e-01</td>
</tr>
</tbody>
</table>

The approximate value functions produced are also the same

Rather than discuss this further, let's go ahead and use our code to generate some results

### 6.4.5 Analysis

Now that our routines are working, let's inspect the solutions

Well start with the following parameterization

```python
def analysis_plot(c=1.25, L0=25, L1=25, a0=2.5, b0=2.0, a1=2.0, b1=2.5, m=25):
    '''
    c: Cost of another draw
    L0: Cost of selecting x0 when x1 is true
    L1: Cost of selecting x1 when x0 is true
    a0, b0: Parameters for f0 (beta distribution)
    a1, b1: Parameters for f1 (beta distribution)
    m: Size of grid
    '''
    f0 = np.clip(st.beta.pdf(np.linspace(0, 1, m), a=a0, b=b0), 1e-6, np.inf)
f0 = f0 / np.sum(f0)
f1 = np.clip(st.beta.pdf(np.linspace(0, 1, m), a=a1, b=b1), 1e-6, np.inf)
f1 = f1 / np.sum(f1)  # Make sure sums to 1

    # Create an instance of our WaldFriedman class
    wf = WaldFriedman(c, L0, L1, f0, f1, m=m)
    # Solve using qe's `compute_fixed_point` function
```
J = qe.compute_fixed_point(wf.bellman_operator, np.zeros(m),
    error_tol=1e-7, verbose=False,
    print_skip=10, max_iter=500)

lb, ub = wf.find_cutoff_rule(J)

# Get draws
ndraws = 500
cdist, tdist = wf.stopping_dist(ndraws=ndraws)

fig, ax = plt.subplots(2, 2, figsize=(12, 9))

ax[0, 0].plot(f0, label="$f_0$"
ax[0, 0].plot(f1, label="$f_1$"
ax[0, 0].set(ylabel="probability of $z_k$", xlabel="$k$", title="Distributions")
ax[0, 0].legend()

ax[0, 1].plot(wf.pgrid, J)
ax[0, 1].annotate(r"$eta$", xy=(lb + 0.025, 0.5), size=14)
ax[0, 1].annotate(r"$\alpha$", xy=(ub + 0.025, 0.5), size=14)
ax[0, 1].vlines(lb, 0.0, wf.payoff_choose_f1(lb), linestyle="--")
ax[0, 1].vlines(ub, 0.0, wf.payoff_choose_f0(ub), linestyle="--")
ax[0, 1].set(ylim=(0, 0.5 * max(L0, L1)), ylabel="cost",
xlabel="$p_k$", title="Value function $J$")

# Histogram the stopping times
ax[1, 0].hist(tdist, bins=np.max(tdist))
ax[1, 0].set_title(f"Stopping times over \(\text{ndraws}\) replications")
ax[1, 0].set(xlabel="time", ylabel="number of stops")
ax[1, 0].annotate(f"mean = {np.mean(tdist)}", xy=(max(tdist) / 2,
    max(np.histogram(tdist, bins=max(tdist))[0]) / 2))

ax[1, 1].hist(cdist, bins=2)
ax[1, 1].set_title(f"Correct decisions over \(\text{ndraws}\) replications")
ax[1, 1].annotate(f"% correct = {np.mean(cdist)}",
    xy=(0.05, ndraws / 2))

plt.tight_layout()
plt.show()

analysis_plot()

Here's a plot of some objects well discuss one by one
The code to generate this figure can be found in wald_solution_plots.py

Value Function

In the top left subfigure we have the two beta distributions, $f_0$ and $f_1$

In the top right we have corresponding value function $J$

It equals $pL_1$ for $p \leq \beta$, and $(1 - p)L_0$ for $p \geq \alpha$

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 $p \in (\beta, \alpha)$

The decision maker continues to sample until the probability that he attaches to model $f_0$ falls below $\beta$ or above $\alpha$

Simulations

The bottom two subfigures show 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.

**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?

```python
analysis_plot(c=2.5)
```

Here's the figure:

---

6.4. A Problem that Stumped Milton Friedman
Notice what happens
The stopping times dropped dramatically!
Increased cost per draw has induced the decision maker usually to take only 1 or 2 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

**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

**6.4.6 Comparison with Neyman-Pearson formulation**

For several reasons, it is useful to describe the theory underlying the test that Navy Captain G. S. Schuyler had been told to use and that led him to approach Milton Friedman and Allan Wallis to convey his conjecture that superior practical procedures existed

Evidently, the Navy had told Captain Schuyler to use what it knew to be a state-of-the-art Neyman-Pearson test

We rely on Abraham Walds [Wal47] elegant summary of Neyman-Pearson theory

For our purposes, watch for these features of the setup:

- the assumption of a fixed sample size $n$
- the application of laws of large numbers, conditioned on alternative probability models, to interpret the probabilities $\alpha$ and $\beta$ 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 $\beta$ and $\alpha$ characterize cut-off rules used to determine $n$ as a random variable
- Laws of large numbers make no appearances in the sequential construction
In chapter 1 of *Sequential Analysis* [Wal47] Abraham Wald summarizes the Neyman-Pearson approach to hypothesis testing.

Wald frames the problem as making a decision about a probability distribution that is partially known.

(You have to assume that *something* is already known in order to state a well posed problem. Usually, *something* means a lot.)

By limiting what is unknown, Wald uses the following simple structure to illustrate the main ideas.

- a decision maker wants to decide which of two distributions $f_0, f_1$ govern an i.i.d. random variable $z$
- The null hypothesis $H_0$ is the statement that $f_0$ governs the data.
- The alternative hypothesis $H_1$ is the statement that $f_1$ governs the data.
- The problem is to devise and analyze a test of hypothesis $H_0$ against the alternative hypothesis $H_1$ on the basis of a sample of a fixed number $n$ independent observations $z_1, z_2, \ldots, z_n$ of the random variable $z$.

To quote Abraham Wald,

- A test procedure leading to the acceptance or rejection of the hypothesis in question is simply a rule specifying, for each possible sample of size $n$, whether the 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 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 $\alpha$ and the probability of an error of the second kind by $\beta$.

Let's listen carefully to how Wald applies a law of large numbers to interpret $\alpha$ and $\beta$:

- The probabilities $\alpha$ and $\beta$ 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 $\alpha$. If $H_1$ is true, the probability is nearly 1 that the proportion of
wrong statements will be approximately $\beta$. Thus, we can say that in the long run [here Wald applies a law of large numbers by driving $M \to \infty$ (our comment, not Walds)] the proportion of wrong statements will be $\alpha$ if $H_0$ is true and $\beta$ if $H_1$ is true.

The quantity $\alpha$ 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 $\alpha$ and $\beta$. Although either $\alpha$ or $\beta$ can be made arbitrarily small by a proper choice of the critical region $W$, it is possible to make both $\alpha$ and $\beta$ arbitrarily small for a fixed value of $n$, i.e., a fixed sample size.

Wald summarizes Neyman and Pearsons setup as follows:

- Neyman and Pearson show that a region consisting of all samples $(z_1, z_2, \ldots, z_n)$ which satisfy the inequality

$$\frac{f_1(z_1) \cdots f_1(z_n)}{f_0(z_1) \cdots f_1(z_n)} \geq k$$

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 $\alpha$.

Wald goes on to discuss Neyman and Pearsons 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 decisions 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 decisions 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.

### 6.5 Job Search III: Search with Learning

**Contents**

- **Job Search III: Search with Learning**
  - Overview
  - Model
  - Take 1: Solution by VFI
6.5.1 Overview

In this lecture we consider an extension of the previously studied job search model of McCall [McC70].

In the McCall model, an unemployed worker decides when to accept a permanent position at a specified wage, given

- his or her discount rate
- the level of unemployment compensation
- the distribution from which wage offers are drawn

In the version considered below, the wage distribution is unknown and must be learned

- The following is based on the presentation in [LS18], section 6.6

Model features

- Infinite horizon dynamic programming with two states and one binary control
- Bayesian updating to learn the unknown distribution

6.5.2 Model

Let's first review the basic McCall model [McC70] and then add the variation we want to consider

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 reconsider next period

The wage sequence $\{W_t\}$ is iid and generated from known density $h$

The worker aims to maximize the expected discounted sum of earnings $E \sum_{t=0}^{\infty} \beta^t y_t$. The function $V$ satisfies the recursion...
\[ V(w) = \max \left\{ \frac{w}{1 - \beta}, c + \beta \int V(w') h(w') dw' \right\} \tag{6.19} \]

The optimal policy has the form \( 1\{w \geq \bar{w}\} \), where \( \bar{w} \) is a constant depending called the reservation wage.

### Offer Distribution Unknown

Now let's extend the model by considering the variation presented in [LS18], section 6.6.

The model is as above, apart from the fact that

- the density \( h \) is unknown
- the worker learns about \( h \) 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 \( h \) 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 \( \pi_0 \) on \( f \) being chosen.

Update rule: workers time \( t \) estimate of the distribution is \( \pi_t f + (1 - \pi_t) g \), where \( \pi_t \) updates via

\[ \pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t) g(w_{t+1})} \tag{6.20} \]

This last expression follows from Bayes rule, which tells us that:

\[ \mathbb{P}\{h = f \mid W = w\} = \frac{\mathbb{P}\{W = w \mid h = f\} \mathbb{P}\{h = f\}}{\mathbb{P}\{W = w\}} \quad \text{and} \quad \mathbb{P}\{W = w\} = \sum_{\psi \in \{f, g\}} \mathbb{P}\{W = w \mid h = \psi\} \mathbb{P}\{h = \psi\} \]

The fact that (6.20) is recursive allows us to progress to a recursive solution method.

Letting

\[ h_\pi(w) := \pi f(w) + (1 - \pi) g(w) \quad \text{and} \quad q(w, \pi) := \frac{\pi f(w)}{\pi f(w) + (1 - \pi) g(w)} \]

we can express the value function for the unemployed worker recursively as follows

\[ V(w, \pi) = \max \left\{ \frac{w}{1 - \beta}, c + \beta \int V(w', \pi') h_\pi(w') dw' \right\} \quad \text{where} \quad \pi' = q(w', \pi) \tag{6.21} \]

Notice that the current guess \( \pi \) is a state variable, since it affects the workers perception of probabilities for future rewards.
Parameterization

Following section 6.6 of [LS18], our baseline parameterization will be

- $f$ is Beta(1, 1) scaled (i.e., draws are multiplied by) some factor $w_m$
- $g$ is Beta(3, 1.2) scaled (i.e., draws are multiplied by) the same factor $w_m$
- $\beta = 0.95$ and $c = 0.6$

With $w_m = 2$, the densities $f$ and $g$ have the following shape

```python
from scipy.stats import beta
import matplotlib.pyplot as plt
import numpy as np

w_m = 2  # Scale factor
x = np.linspace(0, w_m, 200)
plt.figure(figsize=(10, 8))
plt.plot(x, beta.pdf(x, 1, 1, scale=w_m), label='$f$', lw=2)
plt.plot(x, beta.pdf(x, 3, 1.2, scale=w_m), label='$g$', lw=2)
plt.xlim(0, w_m)
plt.legend()
plt.show()
```
Looking Forward

What kind of optimal policy might result from (6.21) 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 $\pi$ in fact it should be decreasing in $\pi$ because

- $f$ is a less attractive offer distribution than $g$
- larger $\pi$ means more weight on $f$ and less on $g$

Thus larger $\pi$ depresses the workers assessment of her future prospects, and relatively low current offers become more attractive.

**Summary:** We conjecture that the optimal policy is of the form $\mathbb{I}\{w \geq \bar{w}(\pi)\}$ for some decreasing function $\bar{w}$. 

542 Chapter 6. Dynamic Programming
6.5.3 Take 1: Solution by VFI

Let's set about solving the model and see how our results match with our intuition.

We begin by solving via value function iteration (VFI), which is natural but ultimately turns out to be second best.

The code is as follows:

```python
from scipy.interpolate import LinearNDInterpolator
from scipy.integrate import fixed_quad
from numpy import maximum as npmax

class SearchProblem:
    """
    A class to store a given parameterization of the "offer distribution unknown" model.

    Parameters
    ----------
    \(\beta\) : scalar(float), optional(default=0.95)
        The discount parameter
    \(c\) : scalar(float), optional(default=0.6)
        The unemployment compensation
    \(F_a\) : scalar(float), optional(default=1)
        First parameter of \(\beta\) distribution on \(F\)
    \(F_b\) : scalar(float), optional(default=1)
        Second parameter of \(\beta\) distribution on \(F\)
    \(G_a\) : scalar(float), optional(default=3)
        First parameter of \(\beta\) distribution on \(G\)
    \(G_b\) : scalar(float), optional(default=1.2)
        Second parameter of \(\beta\) distribution on \(G\)
    \(w_{\text{max}}\) : scalar(float), optional(default=2)
        Maximum wage possible
    \(w_{\text{grid size}}\) : scalar(int), optional(default=40)
        Size of the grid on wages
    \(\pi_{\text{grid size}}\) : scalar(int), optional(default=40)
        Size of the grid on probabilities

    Attributes
    ----------
    \(\beta\), \(c\), \(w_{\text{max}}\) : see Parameters
    \(w_{\text{grid}}\) : np.ndarray
        Grid points over wages, ndim=1
    \(\pi_{\text{grid}}\) : np.ndarray
        Grid points over \(\pi\), ndim=1
    \(\text{grid points}\) : np.ndarray
        Combined grid points, ndim=2
    \(F\) : scipy.stats._distn_infrastructure.rv_frozen
        Beta distribution with params \((F_a, F_b)\), scaled by \(w_{\text{max}}\)
    \(G\) : scipy.stats._distn_infrastructure.rv_frozen
        Beta distribution with params \((G_a, G_b)\), scaled by \(w_{\text{max}}\)
    \(f\) : function
    """
```

6.5. Job Search III: Search with Learning
```python

def __init__(self, β=0.95, c=0.6, F_a=1, F_b=1, G_a=3, G_b=1.2, w_max=2, w_grid_size=40, π_grid_size=40):
    self.β, self.c, self.w_max = β, c, w_max
    self.F = beta(F_a, F_b, scale=w_max)
    self.G = beta(G_a, G_b, scale=w_max)
    self.f, self.g = self.F.pdf, self.G.pdf # Density functions
    self.π_min, self.π_max = 1e-3, 1 - 1e-3 # Avoids instability
    self.π_grid = np.linspace(self.π_min, self.π_max, π_grid_size)
    x, y = np.meshgrid(self.w_grid, self.π_grid)
    self.grid_points = np.column_stack((x.ravel(), y.ravel()))


def q(self, w, π):
    """
    Updates π using Bayes' rule and the current wage observation w.
    """
    new_π = 1.0 / (1 + ((1 - π) * self.g(w)) / (π * self.f(w)))
    # Return new_π when in [π_min, π_max] and else end points
    new_π = np.maximum(np.minimum(new_π, self.π_max), self.π_min)
    return new_π


def bellman_operator(self, v):
    """
    The Bellman operator. Including for comparison. Value function
    iteration is not recommended for this problem. See the
    reservation wage operator below.
    """
    Parameters
    --------
    v : array_like(float, ndim=1, length=len(π_grid))
        An approximate value function represented as a
```
one-dimensional array.

Returns
-------
new_v : array_like(float, ndim=1, length=len(π_grid))
   The updated value function

""
# == Simplify names == #
f, g, β, c, q = self.f, self.g, self.β, self.c, self.q
vf = LinearNDInterpolator(self.grid_points, v)
N = len(v)
new_v = np.empty(N)

for i in range(N):
    w, π = self.grid_points[i, :]
    v1 = w / (1 - β)
    integrand = lambda m: vf(m, q(m, π)) * (π * f(m) + (1 - π) * g(m))
    integral, error = fixed_quad(integrand, 0, self.w_max)
    v2 = c + β * integral
    new_v[i] = max(v1, v2)

return new_v

def get_greedy(self, v):
   ""
   Compute optimal actions taking v as the value function.

Parameters
----------
v : array_like(float, ndim=1, length=len(π_grid))
   An approximate value function represented as a one-dimensional array.

Returns
-------
policy : array_like(float, ndim=1, length=len(π_grid))
   The decision to accept or reject an offer where 1 indicates accept and 0 indicates reject

""
# == Simplify names == #
f, g, β, c, q = self.f, self.g, self.β, self.c, self.q
vf = LinearNDInterpolator(self.grid_points, v)
N = len(v)
policy = np.zeros(N, dtype=int)

for i in range(N):
    w, π = self.grid_points[i, :]
    v1 = w / (1 - β)
```python
integrand = lambda m: vf(m, q(m, π)) * (π * f(m) + (1 - π) * g(m))
integral, error = fixed_quad(integrand, 0, self.w_max)
v2 = c + β * integral
policy[i] = v1 > v2  # Evaluates to 1 or 0

return policy

def res_wage_operator(self, ):
    """
    Updates the reservation wage function guess via the operator Q.
    Parameters
    ----------
    : array_like(float, ndim=1, length=len(π_grid))
        This is reservation wage guess
    Returns
    -------
    new_ : array_like(float, ndim=1, length=len(π_grid))
        The updated reservation wage guess.
    """
    # == Simplify names == #
    β, c, f, g, q = self.β, self.c, self.f, self.g, self.q
    # == Turn into a function == #
    _f = lambda p: np.interp(p, self.π_grid, )

    new_ = np.empty(len())
    for i, π in enumerate(self.π_grid):
        def integrand(x):
            """Integral expression on right-hand side of operator"
            return npmax(x, _f(q(x, π))) * (π * f(x) + (1 - π) * g(x))
        integral, error = fixed_quad(integrand, 0, self.w_max)
        new_[i] = (1 - β) * c + β * integral

    return new_
```

The class `SearchProblem` is used to store parameters and methods needed to compute optimal actions. The Bellman operator is implemented as the method `.bellman_operator()`, while `.get_greedy()` computes an approximate optimal policy from a guess v of the value function. We will omit a detailed discussion of the code because there is a more efficient solution method. These ideas are implemented in the `.res_wage_operator()` method. Before explaining it lets look at solutions computed from value function iteration. Heres the value function:
from mpl_toolkits.mplot3d.axes3d import Axes3D
from matplotlib import cm
from quantecon import compute_fixed_point

sp = SearchProblem(w_grid_size=100, \( \pi \)_grid_size=100)
v_init = np.zeros(len(sp.grid_points)) + sp.c / (1 - sp.\( \beta \))
v = compute_fixed_point(sp.bellman_operator, v_init)
policy = sp.get_greedy(v)

# Make functions from these arrays by interpolation
vf = LinearNDInterpolator(sp.grid_points, v)
pf = LinearNDInterpolator(sp.grid_points, policy)

\( \pi \)_plot_grid_size, w_plot_grid_size = 100, 100
\( \pi \)_plot_grid = np.linspace(0.001, 0.99, \( \pi \)_plot_grid_size)
w_plot_grid = np.linspace(0, sp.w_max, w_plot_grid_size)

Z = np.empty((w_plot_grid_size, \( \pi \)_plot_grid_size))
for i in range(w_plot_grid_size):
    for j in range(\( \pi \)_plot_grid_size):
        Z[i, j] = vf(w_plot_grid[i], \( \pi \)_plot_grid[j])
fig, ax = plt.subplots(figsize=(6, 6))
ax.contourf(\( \pi \)_plot_grid, w_plot_grid, Z, 12, alpha=0.6, cmap=cm.jet)
cs = ax.contour(\( \pi \)_plot_grid, w_plot_grid, Z, 12, colors="black")
ax.clabel(cs, \$\pi\$, fontsize=10)
ax.set_xlabel('\$\pi\$', fontsize=14, rotation=0, labelpad=15)
ax.set_ylabel('\$w\$', fontsize=14, rotation=0, labelpad=15)
plt.show()
The optimal policy:

```python
Z = np.empty((w_plot_grid_size, \pi_plot_grid_size))
for i in range(w_plot_grid_size):
    for j in range(\pi_plot_grid_size):
        Z[i, j] = pf(w_plot_grid[i], \pi_plot_grid[j])

fig, ax = plt.subplots(figsize=(6, 6))
ax.contourf(\pi_plot_grid, w_plot_grid, Z, 1, alpha=0.6, cmap=cm.jet)
ax.contour(\pi_plot_grid, w_plot_grid, Z, 1, colors="black")
ax.set_xlabel('\pi', fontsize=14)
ax.set_ylabel('$w$', fontsize=14, rotation=0, labelpad=15)
ax.text(0.4, 1.0, 'reject')
ax.text(0.7, 1.8, 'accept')
plt.show()
```
The code takes several minutes to run.

The results fit well with our intuition from section *looking forward*

- The black line in the figure above corresponds to the function $\bar{w}(\pi)$ introduced there
- It is decreasing as expected

### 6.5.4 Take 2: A More Efficient Method

Our implementation of VFI can be optimized to some degree.

But instead of pursuing that, 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.

**Another Functional Equation**

To begin, note that when $w = \bar{w}^∗$ (π), the worker is indifferent between accepting and rejecting. Hence the two choices on the right-hand side of (6.21) have equal value:

$$\frac{\bar{w}^∗}{1 - \beta} = c + \beta \int V(w', \pi') h_{\pi}(w') dw'$$

(6.22)

Together, (6.21) and (6.22) give

$$V(w, \pi) = \max \left\{ \frac{w}{1 - \beta}, \frac{\bar{w}^∗}{1 - \beta} \right\}$$

(6.23)

Combining (6.22) and (6.23), we obtain

$$\frac{\bar{w}^∗}{1 - \beta} = c + \beta \int \max \left\{ \frac{w'}{1 - \beta}, \frac{\bar{w}^∗'}{1 - \beta} \right\} h_{\pi}(w') dw'$$

Multiplying by $1 - \beta$, substituting in $\pi' = q(w', \pi)$ and using $\circ$ for composition of functions yields

$$\bar{w}^∗(\pi) = (1 - \beta)c + \beta \int \max \left\{ w', \bar{w}^∗ \circ q(w', \pi) \right\} h_{\pi}(w') dw'$$

(6.24)

Equation (6.24) can be understood as a functional equation, where $\bar{w}^∗$ is the unknown function.

- Lets call it the *reservation wage functional equation* (RWFE).
- The solution $\bar{w}^∗$ to the RWFE is the object that we wish to compute.

**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 $[0, 1]$
- $\|\psi\| := \sup_{x \in [0, 1]} |\psi(x)|$

Consider the operator $Q$ mapping $\psi \in b[0, 1]$ into $Q\psi \in b[0, 1]$ via

$$(Q\psi)(\pi) = (1 - \beta)c + \beta \int \max \left\{ w', \psi \circ q(w', \pi) \right\} h_{\pi}(w') dw'$$

(6.25)

Comparing (6.24) and (6.25), 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 (6.24) and vice versa

Moreover, for any \( \psi, \phi \in b[0, 1] \), basic algebra and the triangle inequality for integrals tells us that

\[
| (Q\psi(\pi)) - (Q\phi(\pi)) | \leq \beta \int \left| \max \left\{ w', \psi \circ q(w', \pi) \right\} - \max \left\{ w', \phi \circ q(w', \pi) \right\} \right| h_{\pi}(w') \, dw' \tag{6.26}
\]

Working case by case, it is easy to check that for real numbers \( a, b, c \) we always have

\[
| \max \{a, b\} - \max \{a, c\} | \leq |b - c| \tag{6.27}
\]

Combining (6.26) and (6.27) yields

\[
| (Q\psi(\pi)) - (Q\phi(\pi)) | \leq \beta \int \left| \psi \circ q(w', \pi) - \phi \circ q(w', \pi) \right| h_{\pi}(w') \, dw' \leq \beta \| \psi - \phi \| \tag{6.28}
\]

Taking the supremum over \( \pi \) now gives us

\[
\| Q\psi - Q\phi \| \leq \beta \| \psi - \phi \| \tag{6.29}
\]

In other words, \( Q \) is a contraction of modulus \( \beta \) on the complete metric space \( (b[0, 1], \| \cdot \|) \).

Hence

• A unique solution \( \bar{w} \) to the RWFE exists in \( b[0, 1] \)

• \( Q^k \psi \to \bar{w} \) uniformly as \( k \to \infty \), for any \( \psi \in b[0, 1] \)

**Implementation**

These ideas are implemented in the `.res_wage_operator()` method from `odu.py` as shown above.

The method corresponds to action of the operator \( Q \).

The following exercise asks you to exploit these facts to compute an approximation to \( \bar{w} \)

**6.5.5 Exercises**

**Exercise 1**

Use the default parameters and the `.res_wage_operator()` method to compute an optimal policy.

Your result should coincide closely with the figure for the optimal policy shown above.

Try experimenting with different parameters, and confirm that the change in the optimal policy coincides with your intuition.
6.5.6 Solutions

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 much shorter than that of the value function approach in *odu_vfi.py*

```python
sp = SearchProblem(_grid_size=50)
_init = np.ones(len(sp._grid))
w_bar = compute_fixed_point(sp.res_wage_operator, _init)

fig, ax = plt.subplots(figsize=(9, 7))
ax.plot(sp._grid, w_bar, linewidth=2, color='black')
ax.set_ylim(0, 2)
ax.grid(axis='x', linewidth=0.25, linestyle='--', color='0.25')
ax.grid(axis='y', linewidth=0.25, linestyle='--', color='0.25')
ax.fill_between(sp._grid, 0, w_bar, color='blue', alpha=0.15)
ax.fill_between(sp._grid, w_bar, 2, color='green', alpha=0.15)
ax.text(0.42, 1.2, 'reject')
ax.text(0.7, 1.8, 'accept')
plt.show()
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Distance</th>
<th>Elapsed (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2.829e-02</td>
<td>1.071e-01</td>
</tr>
<tr>
<td>10</td>
<td>5.174e-03</td>
<td>2.177e-01</td>
</tr>
<tr>
<td>15</td>
<td>9.652e-04</td>
<td>3.220e-01</td>
</tr>
</tbody>
</table>

Converged in 15 steps
6.5.7 Appendix

The next piece of code is just 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.

The code takes a few minutes to run.

```python
# Set up model and compute the function w_bar
sp = SearchProblem(\pi\_grid\_size=50, F\_a=1, F\_b=1)
\pi\_grid, f, g, F, G = sp.\pi\_grid, sp.f, sp.g, sp.F, sp.G
_init = np.ones(len(sp.\pi\_grid))
w_bar_vals = compute_fixed_point(sp.res\_wage\_operator, _init)
w_bar = lambda x: np.interp(x, \pi\_grid, w_bar_vals)

class Agent:
```

6.5. Job Search III: Search with Learning
Holds the employment state and beliefs of an individual agent.

```python
def __init__(self, π=1e-3):
    self.π = π
    self.employed = 1

def update(self, H):
    "Update self by drawing wage offer from distribution H."
    if self.employed == 0:
        w = H.rvs()
        if w >= w_bar(self.π):
            self.employed = 1
        else:
            self.π = 1.0 / (1 + ((1 - self.π) + g(w)) / (self.π + f(w)))
```

```python
num_agents = 5000
separation_rate = 0.025  # Fraction of jobs that end in each period
separation_num = int(num_agents * separation_rate)
agent_indices = list(range(num_agents))
agents = [Agent() for i in range(num_agents)]
sim_length = 600
H = G  # Start with distribution G
change_date = 200  # Change to F after this many periods

unempl_rate = []
for i in range(sim_length):
    if i % 20 == 0:
        print(f"date = (i)"")
    if i == change_date:
        H = F
        # Randomly select separation_num agents and set employment status to 0
        np.random.shuffle(agent_indices)
        separation_list = agent_indices[:separation_num]
        for agent_index in separation_list:
            agents[agent_index].employed = 0
        # Update agents
        for agent in agents:
            agent.update(H)
        unempl_rate = [agent.employed for agent in agents]
        unempl_rate.append(1 - np.mean(employed))

fig, ax = plt.subplots(figsize=(9, 7))
ax.plot(unempl_rate, lw=2, alpha=0.8, label='unemployment rate')
ax.axvline(change_date, color="red")
ax.legend()
plt.show()
```

```
<table>
<thead>
<tr>
<th>Iteration</th>
<th>Distance</th>
<th>Elapsed (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.829e-02</td>
<td>1.090e-01</td>
</tr>
</tbody>
</table>
```
10  5.174e-03  2.204e-01
15  9.652e-04  3.433e-01
Converged in 15 steps

date = 0
date = 20
date = 40
date = 60
date = 80
date = 100
date = 120
date = 140
date = 160
date = 180
date = 200
date = 220
date = 240
date = 260
date = 280
date = 300
date = 320
date = 340
date = 360
date = 380
date = 400
date = 420
date = 440
date = 460
date = 480
date = 500
date = 520
date = 540
date = 560
date = 580
6.6 Job Search IV: Modeling Career Choice

Next we study a computational problem concerning career and job choices.
The model is originally due to Derek Neal [Nea99]
This exposition draws on the presentation in [LS18], section 6.5

**Model features**

- Career and job within career both chosen to maximize expected discounted wage flow
- Infinite horizon dynamic programming with two state variables

### 6.6.2 Model

In what follows we distinguish between a career and a job, where

- a *career* is understood to be a general field encompassing many possible jobs, and
- a *job* is understood to be a position with a particular firm

For workers, wages can be decomposed into the contribution of job and career

\[ w_t = \theta_t + \epsilon_t, \]

where

- \( \theta_t \) is contribution of career at time \( t \)
- \( \epsilon_t \) is contribution of job at time \( t \)

At the start of time \( t \), a worker has the following options

- retain a current (career, job) pair \((\theta_t, \epsilon_t)\) referred to hereafter as stay put
- retain a current career \( \theta_t \) but redraw a job \( \epsilon_t \) referred to hereafter as new job
- redraw both a career \( \theta_t \) and a job \( \epsilon_t \) referred to hereafter as new life

Draws of \( \theta \) and \( \epsilon \) 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

\[
\mathbb{E} \sum_{t=0}^{\infty} \beta^t w_t
\]

subject to the choice restrictions specified above

Let \( V(\theta, \epsilon) \) denote the value function, which is the maximum of (6.30) over all feasible (career, job) policies, given the initial state \((\theta, \epsilon)\)
The value function obeys

\[ V(\theta, \epsilon) = \max\{I, II, III\}, \]

where

\[ I = \theta + \epsilon + \beta V(\theta, \epsilon) \]
\[ II = \theta + \int \epsilon' G(\delta\epsilon') + \beta \int V(\theta, \epsilon') G(\delta\epsilon') \]
\[ III = \int \theta' F(\delta\theta') + \int \epsilon' G(\delta\epsilon') + \beta \int \int V(\theta', \epsilon') G(\delta\epsilon') F(\delta\theta') \]

Evidently I, II and III correspond to stay put, new job and new life, respectively.

**Parameterization**

As in [LS18], section 6.5, we will focus on a discrete version of the model, parameterized as follows:

- both \( \theta \) and \( \epsilon \) take values in the set \( \text{np.linspace}(0, B, N) \) an even grid of \( N \) points between 0 and \( B \) inclusive
  - \( N = 50 \)
  - \( B = 5 \)
  - \( \beta = 0.95 \)

The distributions \( F \) and \( G \) are discrete distributions generating draws from the grid points \( \text{np.linspace}(0, B, N) \)

A very useful family of discrete distributions is the Beta-binomial family, with probability mass function

\[ p(k | n, a, b) = \binom{n}{k} \frac{B(k + a, n - k + b)}{B(a, b)}, \quad k = 0, \ldots, n \]

Interpretation:

- draw \( q \) from a \( \beta \) distribution with shape parameters \( (a, b) \)
- run \( n \) independent binary trials, each with success probability \( q \)
- \( p(k | n, a, b) \) 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

Heres a figure showing the effect of different shape parameters when \( n = 50 \)
from scipy.special import binom, beta
import matplotlib.pyplot as plt
import numpy as np

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()
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()

6.6.3 Implementation: career.py

The code for solving the DP problem described above is found in this file, which is repeated here for convenience

```python
from quantecon.distributions import BetaBinomial

class CareerWorkerProblem:
    
    An instance of the class is an object with data on a particular

6.6. Job Search IV: Modeling Career Choice
problem of this type, including probabilities, discount factor and sample space for the variables.

Parameters
----------

\( \beta \): scalar(float), optional(default=5.0)
  Discount factor

\( B \): scalar(float), optional(default=0.95)
  Upper bound of for both \( \theta \) and \( \theta \)

\( N \): scalar(int), optional(default=50)
  Number of possible realizations for both \( \theta \) and \( \theta \)

\( F_a \): scalar(int or float), optional(default=1)
  Parameter \( 'a' \) from the career distribution

\( F_b \): scalar(int or float), optional(default=1)
  Parameter \( 'b' \) from the career distribution

\( G_a \): scalar(int or float), optional(default=1)
  Parameter \( 'a' \) from the job distribution

\( G_b \): scalar(int or float), optional(default=1)
  Parameter \( 'b' \) from the job distribution

Attributes
----------

\( \beta, B, N \): see Parameters

\( \theta \): array_like(float, ndim=1)
  A grid of values from 0 to B

\( \theta \): array_like(float, ndim=1)
  A grid of values from 0 to B

\( F_{\text{probs}} \): array_like(float, ndim=1)
  The probabilities of different values for \( F \)

\( G_{\text{probs}} \): array_like(float, ndim=1)
  The probabilities of different values for \( G \)

\( F_{\text{mean}} \): scalar(float)
  The mean of the distribution for \( F \)

\( G_{\text{mean}} \): scalar(float)
  The mean of the distribution for \( G \)

```python
def __init__(self, B=5.0, \( \beta=0.95 \), N=50, F_a=1, F_b=1, G_a=1, G_b=1):
    self.\( \beta \), self.N, self.B = \beta, N, B
    self.\( \theta \) = np.linspace(0, B, N)  # set of \( \theta \) values
    self.\( \theta \) = np.linspace(0, B, N)  # set of \( \theta \) values
    self.F_{\text{probs}} = BetaBinomial(N-1, F_a, F_b).pdf()
    self.G_{\text{probs}} = BetaBinomial(N-1, G_a, G_b).pdf()
    self.F_{\text{mean}} = np.sum(self.\( \theta \) * self.F_{\text{probs}})
    self.G_{\text{mean}} = np.sum(self.\( \theta \) * self.G_{\text{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
```

560 Chapter 6. Dynamic Programming
```python
def bellman_operator(self, v):
    """
    The Bellman operator for the career / job choice model of Neal.

    Parameters
    ----------
    v : array_like(float)
        A 2D NumPy array representing the value function
        Interpretation: $v[i, j] = v(\theta_i, \epsilon_j)$

    Returns
    -------
    new_v : array_like(float)
        The updated value function $Tv$ as an array of shape $v.shape$
    """
    new_v = np.empty(v.shape)
    for i in range(self.N):
        for j in range(self.N):
            # stay put
            v1 = self.\theta[i] + self.\beta * v[i, j]

            # new job
            v2 = self.\theta[i] + self.G_mean + self.\beta * v[i, :] @ self.G_probs

            # new life
            v3 = self.G_mean + self.F_mean + self.\beta * self.F_probs @ v @ self.G_probs
            new_v[i, j] = max(v1, v2, v3)
    return new_v

def get_greedy(self, v):
    """
    Compute optimal actions taking $v$ as the value function.

    Parameters
    ----------
    v : array_like(float)
        A 2D NumPy array representing the value function
        Interpretation: $v[i, j] = v(\theta_i, \epsilon_j)$

    Returns
    -------
    policy : array_like(float)
        A 2D NumPy array, where $policy[i, j]$ is the optimal action
        at $v(\theta_i, \epsilon_j)$.

        The optimal action is represented as an integer in the set
        1, 2, 3, where 1 = 'stay put', 2 = 'new job' and 3 = 'new life'
    """
    policy = np.empty(v.shape, dtype=int)
```

6.6. Job Search IV: Modeling Career Choice
The code defines

- a class `CareerWorkerProblem` that
  - encapsulates all the details of a particular parameterization
  - implements the Bellman operator $T$

In this model, $T$ is defined by $Tv(\theta, \epsilon) = \max\{I; II; III\}$, where $I$, $II$ and $III$ are as given in (6.31), replacing $V$ with $v$

The default probability distributions in `CareerWorkerProblem` correspond to discrete uniform distributions (see the Beta-binomial figure)

In fact all our default settings correspond to the version studied in [LS18], section 6.5.

Hence we can reproduce figures 6.5.1 and 6.5.2 shown there, which exhibit the value function and optimal policy respectively

Here's the value function

```python
from mpl_toolkits.mplot3d.axes3d import Axes3D
from matplotlib import cm
import quantecon as qe

# === set matplotlib parameters === #
plt.rcParams['axes.xmargin'] = 0
plt.rcParams['axes.ymargin'] = 0
plt.rcParams['patch.force_edgecolor'] = True

# === solve for the value function === #
wp = CareerWorkerProblem()
v_init = np.ones((wp.N, wp.N)) * 100
v = qe.compute_fixed_point(wp.bellman_operator, v_init,
                           max_iter=200, print_skip=25)

# === plot value function === #
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
```
The optimal policy can be represented as follows (see Exercise 3 for code)
Interpretation:

- If both job and career are poor or mediocre, the worker will experiment with 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
6.6.4 Exercises

Exercise 1

Using the default parameterization in the class CareerWorkerProblem, generate and plot typical sample paths for $\theta$ and $\epsilon$ when the worker follows the optimal policy.

In particular, modulo randomness, reproduce the following figure (where the horizontal axis represents time).

![Graph showing sample paths for $\theta$ and $\epsilon$]

Hint: To generate the draws from the distributions $F$ and $G$, use the class `DiscreteRV`.

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

$T^* :=$ the first point in time from which the worker’s job no longer changes
Evidently, the workers job becomes permanent if and only if \((\theta_t, \epsilon_t)\) enters the stay put region of \((\theta, \epsilon)\) space. Letting \(S\) denote this region, \(T^*\) can be expressed as the first passage time to \(S\) under the optimal policy:

\[
T^* := \inf \{t \geq 0 \mid (\theta_t, \epsilon_t) \in S\}
\]

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

**Exercise 3**

As best you can, reproduce *the figure showing the optimal policy*

Hint: The `get_greedy()` method returns a representation of the optimal policy where values 1, 2 and 3 correspond to stay put, new job and new life respectively. Use this and `contourf` from `matplotlib.pyplot` to produce the different shadings.

Now set \(G_a = G_b = 100\) and generate a new figure with these parameters. Interpret.

### 6.6.5 Solutions

```python
from quantecon import compute_fixed_point

**Exercise 1**

Simulate job / career paths

In reading the code, recall that `optimal_policy[i, j] = policy at (\theta_i, \epsilon_j) = either 1, 2 or 3; meaning stay put, new job and new life`

```python
wp = CareerWorkerProblem()
v_init = np.ones((wp.N, wp.N)) * 100
v = compute_fixed_point(wp.bellman_operator, v_init, verbose=False, max_iter=200)
optimal_policy = wp.get_greedy(v)
F = np.cumsum(wp.F_probs)
G = np.cumsum(wp.G_probs)

def gen_path(T=20):
    i = j = 0
    _index = []
    _index = []
    for t in range(T):
        if optimal_policy[i, j] == 1:  # Stay put
            pass
        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))
```

566 Chapter 6. Dynamic Programming
Exercise 2

The median for the original parameterization can be computed as follows
wp = CareerWorkerProblem()
v_init = np.ones((wp.N, wp.N)) * 100
v = compute_fixed_point(wp.bellman_operator, v_init, max_iter=200, print_ 
→skip=25)
optimal_policy = wp.get_greedy(v)
F = np.cumsum(wp.F_probs)
G = np.cumsum(wp.G_probs)

def gen_first_passage_time():
    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

M = 25000  # Number of samples
samples = np.empty(M)
for i in range(M):
    samples[i] = gen_first_passage_time()
print(np.median(samples))

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Distance</th>
<th>Elapsed (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>1.460e+00</td>
<td>2.550e-01</td>
</tr>
<tr>
<td>50</td>
<td>4.050e-01</td>
<td>5.062e-01</td>
</tr>
<tr>
<td>75</td>
<td>1.123e-01</td>
<td>7.576e-01</td>
</tr>
<tr>
<td>100</td>
<td>3.116e-02</td>
<td>1.010e+00</td>
</tr>
<tr>
<td>125</td>
<td>8.644e-03</td>
<td>1.263e+00</td>
</tr>
<tr>
<td>150</td>
<td>2.398e-03</td>
<td>1.515e+00</td>
</tr>
<tr>
<td>168</td>
<td>9.524e-04</td>
<td>1.695e+00</td>
</tr>
<tr>
<td>Converged in 168 steps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To compute the median with $\beta = 0.99$ instead of the default value $\beta = 0.95$, replace $wp = CareerWorkerProblem()$ with $wp = CareerWorkerProblem(\beta=0.99)$ and increase the $max_iter=200$ in $v = compute_fixed_point(...) \text{ to } max_iter=1000$

The medians are subject to randomness, but should be about 7 and 14 respectively

Not surprisingly, more patient workers will wait longer to settle down to their final job

**Exercise 3**

Here's the code to reproduce the original figure
wp = CareerWorkerProblem()
v_init = np.ones((wp.N, wp.N)) * 100
v = compute_fixed_point(wp.bellman_operator, v_init, max_iter=200, print_=skip=25)
optimal_policy = wp.get_greedy(v)

fig, ax = plt.subplots(figsize=(6,6))
tg, eg = np.meshgrid(wp.θ, wp.)
levels=(0.5, 1.5, 2.5, 3.5)
ax.contourf(tg, eg, optimal_policy.T, levels=levels, cmap=cm.winter, alpha=0.5)
ax.contour(tg, eg, optimal_policy.T, colors='k', levels=levels, linewidths=2)
ax.set_xlabel('θ', fontsize=14)
ax.set_ylabel('', fontsize=14)
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()
Now we want to set $G_a = G_b = 100$ and generate a new figure with these parameters.

To do this replace: 

```
wp = CareerWorkerProblem()
```

with

```
wp = CareerWorkerProblem(G_a=100, G_b=100)
```

In the new figure, you will see that the region for which the worker will stay put has grown because the distribution for $\epsilon$ has become more concentrated around the mean, making high-paying jobs less realistic.

### 6.7 Job Search V: On-the-Job Search
6.7.1 Overview

In this section we solve a simple on-the-job search model

• based on [LS18], exercise 6.18, and [Jov79]

Model features

• job-specific human capital accumulation combined with on-the-job search

• infinite horizon dynamic programming with one state variable and two controls

6.7.2 Model

Let

• $x_t$ denote the time-$t$ job-specific human capital of a worker employed at a given firm

• $w_t$ denote current wages

Let $w_t = x_t (1 - s_t - \phi_t)$, where

• $\phi_t$ is investment in job-specific human capital for the current role

• $s_t$ 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 $\pi(s_t) \in [0, 1]$

Value of offer is $U_{t+1}$, where $\{U_t\}$ is iid with common distribution $F$

Worker has the right to reject the current offer and continue with existing job

In particular, $x_{t+1} = U_{t+1}$ if accepts and $x_{t+1} = G(x_t, \phi_t)$ if rejects

Letting $b_{t+1} \in \{0, 1\}$ be binary with $b_{t+1} = 1$ indicating an offer, we can write

$$x_{t+1} = (1 - b_{t+1})G(x_t, \phi_t) + b_{t+1} \max\{G(x_t, \phi_t), U_{t+1}\} \tag{6.31}$$

Agents objective: maximize expected discounted sum of wages via controls $\{s_t\}$ and $\{\phi_t\}$

Taking the expectation of $V(x_{t+1})$ and using (6.31), the Bellman equation for this problem can be written as

6.7. Job Search V: On-the-Job Search 571
\[ V(x) = \max_{s+\phi \leq 1} \left\{ x(1-s-\phi) + \beta(1-\pi(s))V[G(x,\phi)] + \beta\pi(s) \int V[G(x,\phi) \vee u]F(du) \right\}. \] (6.32)

Here nonnegativity of \( s \) and \( \phi \) is understood, while \( a \vee b := \max\{a,b\} \)

**Parameterization**

In the implementation below, we will focus on the parameterization

\[ G(x,\phi) = A(x\phi)^\alpha, \quad \pi(s) = \sqrt{s} \quad \text{and} \quad F = \text{Beta}(2,2) \]

with default parameter values

- \( A = 1.4 \)
- \( \alpha = 0.6 \)
- \( \beta = 0.96 \)

The Beta(2,2) distribution is supported on (0,1). It has a unimodal, symmetric density peaked at 0.5

**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 \( \phi \)
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 \( \phi \) 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 (6.31) 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 $\phi$ dominates expected return from search

Combining these observations gives us two informal predictions:

1. At any given state $x$, the two controls $\phi$ 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

### 6.7.3 Implementation

The following code solves the DP problem described above

```python
import numpy as np
from scipy.integrate import fixed_quad as integrate
from scipy.optimize import minimize
import scipy.stats as stats

= 1e-4  # A small number, used in the optimization routine

class JvWorker:
    r"
    A Jovanovic-type model of employment with on-the-job search. The value function is given by
    .. math::
        V(x) = \max_{\phi, s} w(x, \phi, s)
    for
    .. math::
        w(x, \phi, s) := x(1 - \phi s)
        + \beta (1 - \pi(s)) V(G(x, \phi))
        + \beta \pi(s) E V[ \max(G(x, \phi), U)]
    Here
    = human capital
    s = search effort
    \phi = investment in human capital
    \pi(s) = probability of new offer given search level s
    x(1 - s) = wage
    G(x, \phi) = new human capital when current job retained
    U = RV with distribution $F$ -- new draw of human capital

    Parameters
    ---------
    A : scalar(float), optional(default=1.4)
```

6.7. Job Search V: On-the-Job Search
Parameter in human capital transition function
\( \alpha \) : scalar(float), optional(default=0.6)
Parameter in human capital transition function
\( \beta \) : scalar(float), optional(default=0.96)
Discount factor
grid_size : scalar(int), optional(default=50)
Grid size for discretization
\( G \) : function, optional(default=lambda x, : A * (x * )**\alpha)
Transition function for human capital
\( \pi \) : function, optional(default=sqrt)
Function mapping search effort \( (\mathbb{r} \in (0,1)) \) to probability of getting new job offer
\( F \) : distribution, optional(default=beta(2,2))
Distribution from which the value of new job offers is drawn

Attributes
----------
\( A, \alpha, \beta \) : see Parameters
\( x\_grid \) : array_like(float)
The grid over the human capital

```python
def __init__(self, A=1.4, \alpha=0.6, \beta=0.96, grid_size=50,
              G=None, \pi=np.sqrt, F=stats.beta(2, 2)):
    self.A, self.\alpha, self.\beta = A, \alpha, \beta
    # === set defaults for G, \pi and F ===#
    self.G = G if G is not None else lambda x, : A * (x * )**\alpha
    self.\pi = \pi
    self.F = F
    # === Set up grid over the state space for DP ===#
    # Max of grid is the max of a large quantile value for F and the
    # fixed point \( y = G(y, 1) \).
    grid_max = max(A**((1 / (1 - \alpha))), self.F.ppf(1 - ))
    self.x_grid = np.linspace(, grid_max, grid_size)

def bellman_operator(self, V, brute_force=False, return_policies=False):
    """
    Returns the approximate value function TV by applying the
    Bellman operator associated with the model to the function V.
    Returns TV, or the V-greedy policies s\_policy and \_\_policy when
    return_policies=True. In the function, the array V is replaced below
    with a function Vf that implements linear interpolation over the
    points \( (V(x), x) \) for x in x\_grid.
    Parameters
    ----------
    V : array_like(float)
    """
Array representing an approximate value function
brute_force : bool, optional (default=False)
Default is False. If the brute_force flag is True, then grid
search is performed at each maximization step.
return_policies : bool, optional (default=False)
Indicates whether to return just the updated value function
TV or both the greedy policy computed from V and TV

Returns
-------
s_policy : array_like (float)
The greedy policy computed from V. Only returned if
return_policies == True
new_V : array_like (float)
The updated value function TV, as an array representing the
values TV(x) over x in x_grid.

# === simplify names, set up arrays, etc. === #
Vf = lambda x: np.interp(x, self.x_grid, V)
N = len(self.x_grid)
new_V, s_policy, _policy = np.empty(N), np.empty(N), np.empty(N)
a, b = F.ppf(0.005), F.ppf(0.995) # Quantiles, for integration

c1 = lambda z: 1.0 - sum(z) # used to enforce s + <= 1

c2 = lambda z: z[0] - # used to enforce s >=
c3 = lambda z: z[1] - # used to enforce >=
guess = (0.2, 0.2)
constraints = [{"type": "ineq", "fun": i} for i in [c1, c2, c3]]

# === solve r.h.s. of Bellman equation === #
for i, x in enumerate(self.x_grid):

    # === set up objective function === #
def w(z):
        s, = z
        h = lambda u: Vf(np.maximum(G(x, ), u)) * F.pdf(u)
        integral, err = integrate(h, a, b)
        q = π(s) * integral + (1.0 - π(s)) * Vf(G(x, ))
        # == minus because we minimize == #
        return - x * (1.0 - s) - β * q

    # === either use SciPy solver === #
    if not brute_force:
        max_s, max_ = minimize(w, guess, constraints=constraints,
                                options={"disp": 0},
                                method="COBYLA")["x"]
        max_val = -w((max_s, max_))

    # === or search on a grid === #
else:
    search_grid = np.linspace(1.0, 15)
    max_val = -1.0
    for s in search_grid:
        for s_prime in search_grid:
            current_val = -w((s, s_prime))
            if s + s_prime <= 1.0 else -1.0
            if current_val > max_val:
                max_val, max_s, max_s_prime = current_val, s, s_prime

    # === store results ===#
    new_V[i] = max_val
    s_policy[i], _policy[i] = max_s, max_s_prime

    if return_policies:
        return s_policy, _policy
    else:
        return new_V

The code is written to be relatively generic and hence reusable

- For example, we use generic \( G(x, \phi) \) instead of specific \( A(x, \phi) \)

Regarding the imports

- `fixed_quad` is a simple non-adaptive integration routine
- `fmin_slsqp` is a minimization routine that permits inequality constraints

Next we build a class called `JvWorker` that

- packages all the parameters and other basic attributes of a given model
- implements the method `bellman_operator` for value function iteration

The `bellman_operator` method takes a candidate value function \( V \) and updates it to \( TV \) via

\[
TV(x) = \min_{s+\phi \leq 1} w(s, \phi)
\]

where

\[
w(s, \phi) := -\left\{ x(1-s-\phi) + \beta(1-\pi(s))V[G(x, \phi)] + \beta\pi(s) \int V[G(x, \phi) \lor u]F(du) \right\}
\] (6.33)

Here we are minimizing instead of maximizing to fit with SciPy’s optimization routines

When we represent \( V \), it will be with a NumPy array \( V \) giving values on grid \( x_{\text{grid}} \)

But to evaluate the right-hand side of (6.33), we need a function, so we replace the arrays \( V \) and \( x_{\text{grid}} \) with a function \( Vf \) that gives linear interpolation of \( V \) on \( x_{\text{grid}} \)

Hence in the preliminaries of `bellman_operator`

- from the array \( V \) we define a linear interpolation \( Vf \) of its values
  - \( c1 \) is used to implement the constraint \( s + \phi \leq 1 \)
c2 is used to implement \( s \geq \epsilon \), a numerically stable alternative to the true constraint \( s \geq 0 \)

c3 does the same for \( \phi \)

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 (6.33).

The function is minimized over all feasible \((s, \phi)\) pairs, either by

- a relatively sophisticated solver from SciPy called fmin_slsqp, or
- brute force search over a grid

The former is much faster, but convergence to the global optimum is not guaranteed. Grid search is a simple way to check results.

### 6.7.4 Solving for Policies

Let's plot the optimal policies and see what they look like.

The code is as follows

```python
import matplotlib.pyplot as plt
from quantecon import compute_fixed_point

# === solve for optimal policy === #
wp = JvWorker(grid_size=25)
v_init = wp.x_grid * 0.5
V = compute_fixed_point(wp.bellman_operator, v_init, max_iter=40)
s_policy, _policy = wp.bellman_operator(V, return_policies=True)

# === plot policies ===#
plots = [_policy, s_policy, V]
titles = ["policy", "s policy", "value function"]
fig, axes = plt.subplots(3, 1, figsize=(12, 12))
for ax, plot, title in zip(axes, plots, titles):
    ax.plot(wp.x_grid, plot)
    ax.set(title=title)
    ax.grid()
axes[-1].set_xlabel("x")
plt.show()
```

It produces the following figure.
Fig. 6.2: Optimal policies

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 section

- 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
6.7.5 Exercises

**Exercise 1**

Let's look at the dynamics for the state process \( \{x_t\} \) associated with these policies.

The dynamics are given by (6.31) when \( \phi_t \) and \( s_t \) are chosen according to the optimal policies, and \( P\{b_{t+1} = 1\} = \pi(s_t) \).

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 \( \text{plot}\_\text{grid} \), 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. Set

\[
K = 50 \\
\text{plot}\_\text{grid}\_\text{max}, \text{plot}\_\text{grid}\_\text{size} = 1.2, 100 \\
\text{plot}\_\text{grid} = \text{np.linspace}(0, \text{plot}\_\text{grid}\_\text{max}, \text{plot}\_\text{grid}\_\text{size}) \\
\text{fig}, \text{ax} = \text{plt.subplots()} \\
\text{ax.set_xlim}(0, \text{plot}\_\text{grid}\_\text{max}) \\
\text{ax.set_ylim}(0, \text{plot}\_\text{grid}\_\text{max})
\]

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 \( \phi_t \approx 0.6 \).

**Exercise 2**

In the preceding exercise we found that \( s_t \) converges to zero and \( \phi_t \) converges to about 0.6.

Since these results were calculated at a value of \( \beta \) 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 that the infinitely patient worker does not search in the long run (i.e., \( s_t = 0 \) for large \( t \)).

Thus, given \( \phi \), 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 \( \phi \), and examine the best choice of \( \phi \).

Can you give a rough interpretation for the value that you see?

6.7.6 Solutions

**Exercise 1**

Here's code to produce the 45 degree diagram.
import random

wp = JvWorker(grid_size=25)
G, π, F = wp.G, wp.π, wp.F  # Simplify names

v_init = wp.x_grid * 0.5
print("Computing value function")
V = compute_fixed_point(wp.bellman_operator, v_init, max_iter=40,
    →verbose=False)
print("Computing policy functions")
s_policy, _policy = wp.bellman_operator(V, return_policies=True)

# Turn the policy function arrays into actual functions
s = lambda y: np.interp(y, wp.x_grid, s_policy)
= lambda y: np.interp(y, wp.x_grid, _policy)

def h(x, b, U):
    return (1 - b) * G(x, (x)) + b * max(G(x, (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))
ax.set_xlim(0, plot_grid_max)
ax.set_ylim(0, plot_grid_max)
ticks = (0.25, 0.5, 0.75, 1.0)
ax.set_xticks(ticks, yticks=ticks)
ax.set_xlabel('$x_t$', fontsize=16)
ax.set_ylabel('$x_{t+1}$', fontsize=16, rotation='horizontal')

ax.plot(plot_grid, plot_grid, 'k--')  # 45 degree line
for x in plot_grid:
    for i in range(50):
        b = 1 if random.uniform(0, 1) < π(s(x)) else 0
        U = wp.F.rvs(1)
y = h(x, b, U)
ax.plot(x, y, 'go', alpha=0.25)

plt.show()
Looking at the dynamics, we can see that

- If $x_t$ is below about 0.2 the dynamics are random, but $x_{t+1} > x_t$ is very likely
- As $x_t$ increases the dynamics become deterministic, and $x_t$ converges to a steady state value close to 1

Referring back to the figure here we see that $x_t \approx 1$ means that $s_t = s(x_t) \approx 0$ and $\phi_t = \phi(x_t) \approx 0.6$

**Exercise 2**

The figure can be produced as follows
wp = JvWorker(grid_size=25)

def xbar():
    return (wp.A + **wp.α)**(1 / (1 - wp.α))

_grid = np.linspace(0, 1, 100)
fig, ax = plt.subplots(figsize=(9, 7))
ax.set_xlabel(r'$\phi$', fontsize=16)
ax.plot(_grid, [xbar() * (1 - ) for in _grid], 'b-', label='$w^*(\phi)$')
ax.legend(loc='upper left')
plt.show()

Observe that the maximizer is around 0.6

This this is similar to the long run value for $\phi$ obtained in exercise 1

Hence the behaviour 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
6.8 Optimal Growth I: The Stochastic Optimal Growth Model

6.8.1 Overview

In this lecture we are going to study a simple optimal growth model with one agent. The model is a version of the standard one sector infinite horizon growth model studied in

- [SLP89], chapter 2
- [LS18], section 3.1
- EDTC, chapter 1
- [Sun96], chapter 12

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 will discuss some of the technical details of dynamic programming as we go along.

6.8.2 The Model

Consider an agent who owns an amount $y_t \in \mathbb{R}_+ := [0, \infty)$ of a consumption good at time $t$.

This output can either be consumed or invested.

When the good is invested it is transformed one-for-one into capital.

The resulting capital stock, denoted here by $k_{t+1}$, will then be used for production.

Production is stochastic, in that it also depends on a shock $\xi_{t+1}$ realized at the end of the current period.

Next period output is

$$y_{t+1} := f(k_{t+1})\xi_{t+1}$$

where $f : \mathbb{R}_+ \rightarrow \mathbb{R}_+$ is called the production function.
The resource constraint is

\[ k_{t+1} + c_t \leq y_t \quad (6.34) \]

and all variables are required to be nonnegative.

**Assumptions and Comments**

In what follows,

- The sequence \( \{\xi_t\} \) is assumed to be IID
- The common distribution of each \( \xi_t \) will be denoted \( \phi \)
- 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.

**Optimization**

Taking \( y_0 \) as given, the agent wishes to maximize

\[
E \left[ \sum_{t=0}^{\infty} \beta^t u(c_t) \right] \quad (6.35)
\]

subject to

\[ y_{t+1} = f(y_t - c_t)\xi_{t+1} \quad \text{and} \quad 0 \leq c_t \leq y_t \quad \text{for all} \ t \quad (6.36) \]

where

- \( u \) is a bounded, continuous and strictly increasing utility function and
- \( \beta \in (0, 1) \) is a discount factor

In (6.36) we are assuming that the resource constraint (6.34) holds with equality which is reasonable because \( u \) is strictly increasing and no output will be wasted at the optimum.

In summary, the agents 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 (6.34),
optimal, in the sense that it maximizes (6.35) relative to all other feasible consumption sequences, and
4. adapted, in the sense that the action \( c_t \) depends only on observable outcomes, not future outcomes such as \( \xi_{t+1} \)

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

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

Well be particularly interested in Markov policies, which are maps from the current state \( y_t \) into a current action \( c_t \)

For dynamic programming problems such as this one (in fact for any Markov decision process), the optimal policy is always a Markov policy

In other words, the current state \( y_t \) provides a sufficient statistic for the history in terms of making an optimal decision today

This is quite intuitive but if you wish you can find proofs in texts such as [SLP89] (section 4.1)

Hereafter we focus on finding the best Markov policy

In our context, a Markov policy is a function \( \sigma : \mathbb{R}_+ \rightarrow \mathbb{R}_+ \), 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 \( \sigma \) a feasible consumption policy if it satisfies

\[
0 \leq \sigma(y) \leq y \quad \text{for all } y \in \mathbb{R}_+ \quad (6.37)
\]

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 \( \Sigma \)

Each \( \sigma \in \Sigma \) determines a continuous state Markov process \( \{y_t\} \) for output via

\[
y_{t+1} = f(y_t - \sigma(y_t))\xi_{t+1}, \quad y_0 \text{ given} \quad (6.38)
\]

This is the time path for output when we choose and stick with the policy \( \sigma \)

We insert this process into the objective function to get

\[
\mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(c_t) \right] = \mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(\sigma(y_t)) \right]
\]  
(6.39)

This is the total expected present value of following policy \( \sigma \) 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

**Optimality**

The **policy value function** \( v_{\sigma} \) associated with a given policy \( \sigma \) is the mapping defined by

\[
v_{\sigma}(y) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(\sigma(y_t)) \right]
\]  
(6.40)

when \( \{y_t\} \) is given by (6.38) with \( y_0 = y \)

In other words, it is the lifetime value of following policy \( \sigma \) starting at initial condition \( y \)

The **value function** is then defined as

\[
v^*(y) := \sup_{\sigma \in \Sigma} v_{\sigma}(y)
\]  
(6.41)

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 (6.41) for all \( y \in \mathbb{R}_+ \)

**The Bellman Equation**

With our assumptions on utility and production function, the value function as defined in (6.41) also satisfies a **Bellman equation**

For this problem, the Bellman equation takes the form

\[
w(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y - c)z) \phi(dz) \right\} \quad (y \in \mathbb{R}_+)
\]  
(6.42)

This is a **functional equation in** \( w \)

The term \( \int w(f(y - c)z) \phi(dz) \) can be understood as the expected next period value when

- \( w \) 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, (6.42) holds when $w = v^*$

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

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 $w$ on $\mathbb{R}_+$, we say that $\sigma \in \Sigma$ is $w$-greedy if $\sigma(y)$ is a solution to

$$
\max_{0 \leq c \leq y} \left\{ w(c) + \beta \int w(f(y - c)z)\phi(dz) \right\}
$$

for every $y \in \mathbb{R}_+$

In other words, $\sigma \in \Sigma$ is $w$-greedy if it optimally trades off current and future rewards when $w$ is taken to be the value function

In our setting, we have the following key result

A feasible consumption policy is optimal if and only if it is $v^*$-greedy

The intuition is similar to the intuition for the Bellman equation, which was provided after (6.42)

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

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
In other words, $T$ sends the function $w$ into the new function $Tw$ defined \((6.44)\).

By construction, the set of solutions to the Bellman equation \((6.42)\) exactly coincides with the set of fixed points of $T$.

For example, if $Tw = w$, then, for any $y \geq 0$,

$$w(y) = Tw(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y - c)z) \phi(dz) \right\}$$

which says precisely that $w$ is a solution to the Bellman equation.

It follows that $v^*$ is a fixed point of $T$.

**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 $w$, the sequence $w, Tw, T^2w, \ldots$ 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.

**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.

---

**Chapter 6. Dynamic Programming**
Nevertheless, their main conclusions are usually in line with those stated for the bounded case just above (as long as we drop the word bounded)

Consult, for example, section 12.2 of EDTC, [Kam12] or [MdRV10]

6.8.3 Computation

Let's now look at computing the value function and the optimal policy

Fitted Value Iteration

The first step is to compute the value function by value function iteration

In theory, the algorithm is as follows

1. Begin with a function \( w \) an initial condition
2. Solving (6.44), obtain the function \( Tw \)
3. Unless some stopping condition is satisfied, set \( w = Tw \) and go to step 2

This generates the sequence \( w, Tw, T^2w, \ldots \)

However, there is a problem we must confront before we implement this procedure: The iterates can neither be calculated exactly nor stored on a computer

To see the issue, consider (6.44)

Even if \( w \) is a known function, unless \( Tw \) can be shown to have some special structure, the only way to store it is to record the value \( Tw(y) \) for every \( y \in \mathbb{R}_+ \)

Clearly this is impossible

What we will do instead is use fitted value function iteration

The procedure is to record the value of the function \( Tw \) at only finitely many grid points \( y_1 < y_2 < \cdots < y_I \) and reconstruct it from this information when required

More precisely, the algorithm will be

1. Begin with an array of values \( \{w_1, \ldots, w_I\} \) representing the values of some initial function \( w \) on the grid points \( \{y_1, \ldots, y_I\} \)
2. Build a function \( \hat{w} \) on the state space \( \mathbb{R}_+ \) by interpolation or approximation, based on these data points
3. Obtain and record the value \( T\hat{w}(y_i) \) on each grid point \( y_i \) by repeatedly solving (6.44)
4. Unless some stopping condition is satisfied, set \( \{w_1, \ldots, w_I\} = \{T\hat{w}(y_1), \ldots, T\hat{w}(y_I)\} \) and go to step 2

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 \( Tw \), but also combine well with the broader iteration algorithm described above
One good choice from both respects is continuous piecewise linear interpolation (see this paper for further discussion)

The next figure illustrates piecewise linear interpolation of an arbitrary function on grid points
0, 0.2, 0.4, 0.6, 0.8, 1

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

def f(x):
    y1 = 2 * np.cos(6 * x) + np.sin(14 * x)
    return y1 + 2.5

c_grid = np.linspace(0, 1, 6)

def Af(x):
    return np.interp(x, c_grid, f(c_grid))

f_grid = np.linspace(0, 1, 150)

fig, ax = plt.subplots(figsize=(10, 6))
ax.set(xlim=(0, 1), ylim=(0, 6))
ax.plot(f_grid, f(f_grid), 'b-', lw=2, alpha=0.8, label='true function')
ax.plot(f_grid, Af(f_grid), 'g-', lw=2, alpha=0.8, label='linear approximation')
ax.vlines(c_grid, c_grid * 0, f(c_grid), linestyle='dashed', alpha=0.5)
ax.legend(loc='upper center')
plt.show()
```
Another advantage of piecewise linear interpolation is that it preserves useful shape properties such as monotonicity and concavity / convexity.

**The Bellman Operator**

Here’s a function that implements the Bellman operator using linear interpolation:

```python
from scipy.optimize import fminbound

def bellman_operator(w, grid, β, u, f, shocks, Tw=None, compute_policy=0):
    
    The approximate Bellman operator, which computes and returns the updated value function Tw on the grid points. An array to store the new set of values Tw is optionally supplied (to avoid having to allocate new arrays at each iteration). If supplied, any existing data in Tw will be overwritten.

    Parameters
    ---------
    w : array_like(float, ndim=1)
        The value of the input function on different grid points
    grid : array_like(float, ndim=1)
        The set of grid points
    β : scalar
        The discount factor
    u : function
        The utility function
    f : function
        The production function
    shocks : numpy array
        An array of draws from the shock, for Monte Carlo integration (to compute expectations).
    Tw : array_like(float, ndim=1) optional (default=None)
        Array to write output values to
    compute_policy : Boolean, optional (default=False)
        Whether or not to compute policy function

    
    # === Apply linear interpolation to w === #
    w_func = lambda x: np.interp(x, grid, w)

    # == Initialize Tw if necessary == #
    if Tw is None:
        Tw = np.empty_like(w)

    if compute_policy:
        σ = np.empty_like(w)

    # === set Tw[i] = max_c { u(c) + β E w(f(y - c) z)} == #
    for i, y in enumerate(grid):
```
```python
def objective(c):
    return - u(c) - np.mean(w_func(f(y - c) * shocks))

c_star = fminbound(objective, 1e-10, y)
if compute_policy:
    σ[i] = c_star
    Tw[i] = - objective(c_star)

if compute_policy:
    return Tw, σ
else:
    return Tw
```

The arguments to `bellman_operator` are described in the docstring to the function.

Notice that the expectation in (6.44) is computed via Monte Carlo, using the approximation

\[
\int w(f(y - c)z)\phi(dz) \approx \frac{1}{n} \sum_{i=1}^{n} w(f(y - c)\xi_i)
\]

where \{\xi_i\}_{i=1}^{n} are IID draws from \(\phi\).

Monte Carlo is not always the most efficient way to compute integrals numerically but it does have some theoretical advantages in the present setting.

(For example, it preserves the contraction mapping property of the Bellman operator see, e.g., [PalS13])

**An Example**

Let's test out our operator when

- \(f(k) = k^\alpha\)
- \(u(c) = \ln c\)
- \(\phi\) is the distribution of \(\exp(\mu + \sigma \zeta)\) when \(\zeta\) is standard normal

As is well-known (see [LS18], section 3.1.2), for this particular problem an exact analytical solution is available, with

\[
\nu^*(y) = \frac{\ln(1 - \alpha\beta)}{1 - \beta} + \frac{(\mu + \alpha \ln(\alpha\beta))}{1 - \alpha} \left[ \frac{1}{1 - \beta} - \frac{1}{1 - \alpha\beta} \right] + \frac{1}{1 - \alpha\beta} \ln y \tag{6.45}
\]

The optimal consumption policy is

\[
\sigma^*(y) = (1 - \alpha\beta)y
\]

Let's wrap this model in a class because we'll use it some later lectures too.
class LogLinearOG:
    ""
    Log linear optimal growth model, with log utility, CD production and
    multiplicative lognormal shock, so that

    \[ y = f(k, z) = z k^\alpha \]

    with \( z \sim \text{LN}(\mu, s) \).

    The class holds parameters and true value and policy functions.
    ""

def __init__(self, α=0.4, β=0.96, μ=0, s=0.1):
    self.α, self.β, self.μ, self.s = α, β, μ, s

    # == Some useful constants == #
    self.ab = α * β
    self.c1 = np.log(1 - self.ab) / (1 - β)
    self.c2 = (μ + α * np.log(self.ab)) / (1 - α)
    self.c3 = 1 / (1 - β)
    self.c4 = 1 / (1 - self.ab)

def u(self, c):
    " Utility "
    return np.log(c)

def u_prime(self, c):
    return 1 / c

def f(self, k):
    " Deterministic part of production function. "
    return k**self.α

def f_prime(self, k):
    return self.α * k**(self.α - 1)

def c_star(self, y):
    " True optimal policy. "
    return (1 - self.α * self.β) * y

def v_star(self, y):
    " True value function. "
    return self.c1 + self.c2 * (self.c3 - self.c4) + self.c4 * np.log(y)

A First Test

To test our code, we want to see if we can replicate the analytical solution numerically, using fitted value function iteration

First, having run the code for the log linear model shown above, let's generate an instance
We need a grid and some shock draws for Monte Carlo integration

```
grid_max = 4       # Largest grid point
grid_size = 200    # Number of grid points
shock_size = 250   # Number of shock draws in Monte Carlo integral

grid = np.linspace(1e-5, grid_max, grid_size)
shocks = np.exp(μ + s * np.random.randn(shock_size))
```

Now lets do some tests

As one preliminary test, lets see what happens when we apply our Bellman operator to the exact solution $v^*$

In theory, the resulting function should again be $v^*$

In practice we expect some small numerical error

```
w = bellman_operator(v_star(grid),
                    grid, β,
                    np.log,
                    lambda k: k**α,
                    shocks)

fig, ax = plt.subplots(figsize=(9, 5))
ax.set_yscale('log')
ax.plot(grid, w, lw=2, alpha=0.6, label='$Tv^*$')
ax.plot(grid, v_star(grid), lw=2, alpha=0.6, label='$v^*$')
ax.legend(loc='lower right')
plt.show()
```

Here's the output:
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 will start with is $w(y) = 5 \ln(y)$

```python
w = 5 * np.log(grid)  # An initial condition
n = 35
fig, ax = plt.subplots(figsize=(9, 6))
ax.set_ylim(-40, 10)
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 = bellman_operator(w, grid, beta, np.log, lambda k: k**alpha, shocks)
    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='lower right')
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

We can write a function that iterates until the difference is below a particular tolerance level

```python
def solve_optgrowth(initial_w, tol=1e-6, max_iter=500):
    w = initial_w  # Set initial condition
    error = tol + 1
    i = 0

    # == Create storage array for bellman_operator. Reduces memory allocation and speeds code up == #
    Tw = np.empty(len(grid))

    # Iterate to find solution
    while error > tol and i < max_iter:
        w_new = bellman_operator(w, grid, \beta, 
```

Chapter 6. Dynamic Programming
We can check our result by plotting it against the true value

```python
initial_w = 5 * np.log(grid)
fig, ax = plt.subplots(figsize=(9, 5))
ax.set_ylim(-35, -24)
ax.plot(grid, solve_optgrowth(initial_w), lw=2, alpha=0.6, label='approximate value function')
ax.plot(grid, v_star(grid), lw=2, alpha=0.6, label='true value function')
ax.legend(loc='lower right')
plt.show()
```

Alternatively, we can use QuantEcons `compute_fixed_point` function to converge to $v^*$

```python
from quantecon import compute_fixed_point
initial_w = 5 * np.log(grid)
# Turn the Bellman operator into a function of one variable
```
\( T = \lambda w: \text{bellman_operator}(w, \) \\
grid, \betap\) \\
np.log, \lambda k: k**\alpha, \) \\
shocks, \) \\
compute_policy=False) \)

\( v_{\star \text{approx}} = \text{compute_fixed_point}(T, \text{initial}_w, \) \\
error_tol=1e-5, \) \\
max_iter=500, \) \\
verbose=2, \) \\
print_skip=10, \) \\
method='iteration') \)

Here's the output

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Distance</th>
<th>Elapsed (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>7.052e-01</td>
<td>9.205e-01</td>
</tr>
<tr>
<td>20</td>
<td>4.683e-01</td>
<td>1.875e+00</td>
</tr>
<tr>
<td>30</td>
<td>3.114e-01</td>
<td>2.779e+00</td>
</tr>
<tr>
<td>40</td>
<td>2.070e-01</td>
<td>3.695e+00</td>
</tr>
<tr>
<td>50</td>
<td>1.376e-01</td>
<td>4.594e+00</td>
</tr>
<tr>
<td>60</td>
<td>9.149e-02</td>
<td>5.513e+00</td>
</tr>
<tr>
<td>70</td>
<td>6.083e-02</td>
<td>6.383e+00</td>
</tr>
<tr>
<td>80</td>
<td>4.044e-02</td>
<td>7.315e+00</td>
</tr>
<tr>
<td>90</td>
<td>2.689e-02</td>
<td>8.191e+00</td>
</tr>
<tr>
<td>100</td>
<td>1.787e-02</td>
<td>9.069e+00</td>
</tr>
<tr>
<td>110</td>
<td>1.188e-02</td>
<td>9.981e+00</td>
</tr>
<tr>
<td>120</td>
<td>7.901e-03</td>
<td>1.088e+01</td>
</tr>
<tr>
<td>130</td>
<td>5.253e-03</td>
<td>1.175e+01</td>
</tr>
<tr>
<td>140</td>
<td>3.492e-03</td>
<td>1.261e+01</td>
</tr>
<tr>
<td>150</td>
<td>2.322e-03</td>
<td>1.349e+01</td>
</tr>
<tr>
<td>160</td>
<td>1.544e-03</td>
<td>1.439e+01</td>
</tr>
<tr>
<td>170</td>
<td>1.026e-03</td>
<td>1.533e+01</td>
</tr>
<tr>
<td>180</td>
<td>6.822e-04</td>
<td>1.630e+01</td>
</tr>
<tr>
<td>190</td>
<td>4.536e-04</td>
<td>1.722e+01</td>
</tr>
<tr>
<td>200</td>
<td>3.016e-04</td>
<td>1.816e+01</td>
</tr>
<tr>
<td>210</td>
<td>2.005e-04</td>
<td>1.904e+01</td>
</tr>
<tr>
<td>220</td>
<td>1.333e-04</td>
<td>1.995e+01</td>
</tr>
<tr>
<td>230</td>
<td>8.861e-05</td>
<td>2.085e+01</td>
</tr>
<tr>
<td>240</td>
<td>5.891e-05</td>
<td>2.184e+01</td>
</tr>
<tr>
<td>250</td>
<td>3.917e-05</td>
<td>2.270e+01</td>
</tr>
<tr>
<td>260</td>
<td>2.604e-05</td>
<td>2.355e+01</td>
</tr>
<tr>
<td>270</td>
<td>1.731e-05</td>
<td>2.440e+01</td>
</tr>
<tr>
<td>280</td>
<td>1.151e-05</td>
<td>2.526e+01</td>
</tr>
<tr>
<td>284</td>
<td>9.776e-06</td>
<td>2.561e+01</td>
</tr>
</tbody>
</table>
Converged in 284 steps

Let's have a look at the result
The figure shows that we are pretty much on the money

**The Policy Function**

To compute an approximate optimal policy, we take the approximate value function we just calculated and then compute the corresponding greedy policy.

The next figure compares the result to the exact solution, which, as mentioned above, is $\sigma(y) = (1 - \alpha\beta)y$.
The figure shows that we've done a good job in this instance of approximating the true policy.

### 6.8.4 Exercises

**Exercise 1**

Once an optimal consumption policy $\sigma$ is given, income follows (6.38)

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

```python
s = 0.05
shocks = np.exp(\mu + s * np.random.randn(shock_size))
```

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

### 6.8.5 Solutions

**Exercise 1**

Here’s one solution (assuming as usual that you’ve executed everything above)

```python
def simulate_og(\sigma, y0=0.1, ts_length=100):
    
    'Compute a time series given consumption policy \sigma.
    '
```
```python
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(y[t]))**α * np.exp(μ + s * ξ[t])
return y

fig, ax = plt.subplots(figsize=(9, 6))

for β in (0.8, 0.9, 0.98):
    Tw = np.empty(len(grid))
    initial_w = 5 * np.log(grid)

    v_star_approx = compute_fixed_point(bellman_operator,
                                          initial_w,
                                          1e-5,       # error_tol
                                          500,       # max_iter
                                          False,     # verbose
                                          5,         # print_skip
                                          'iteration',
                                          grid,
                                          β,
                                          np.log,
                                          lambda k: k**α,
                                          shocks,
                                          Tw=Tw,
                                          compute_policy=False)

    Tw, σ = bellman_operator(v_star_approx,
                              grid,
                              β,
                              np.log,
                              lambda k: k**α,
                              shocks,
                              compute_policy=True)

    σ_func = lambda x: np.interp(x, grid, σ)
y = simulate_og(σ_func)
    ax.plot(y, lw=2, alpha=0.6, label=r'\beta = (' + r'{\beta}$\'$)

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

### 6.9 Optimal Growth II: Time Iteration
6.9.1 Overview

In this lecture we 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 will use this structure to obtain an Euler equation based method that is more efficient than value function iteration for this and some other closely related applications.

In a subsequent lecture we will see that the numerical implementation part of the Euler equation method can be further adjusted to obtain even more efficiency.

6.9.2 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
$$v^*(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y - c)z)\phi(dz) \right\} \quad \text{for all } y \in \mathbb{R}_+$$ (6.46)

Let the optimal consumption policy be denoted by $c^*$

We know that $c^*$ is a $v^*$ greedy policy, so that $c^*(y)$ is the maximizer in (6.46)

The conditions above imply that

- $c^*$ 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 $0 < c^*(y) < y$ for all strictly positive $y$, and
- the value function is strictly concave and continuously differentiable, with

$$\left( v^* \right)'(y) = u'(c^*(y)) := (u' \circ c^*)(y)$$ (6.47)

The last result is called the **envelope condition** due to its relationship with the **envelope theorem**

To see why (6.47) 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 (6.46), which is

$$u'(c^*(y)) = \beta \int \left( v^* \right)'(f(y - c^*(y))z)f'(y - c^*(y))z\phi(dz)$$ (6.48)

Combining (6.47) and the first-order condition (6.48) gives the famous **Euler equation**

$$(u' \circ c^*)(y) = \beta \int (u' \circ c^*)(f(y - c^*(y))z)f'(y - c^*(y))z\phi(dz)$$ (6.49)

We can think of the Euler equation as a functional equation

$$(u' \circ \sigma)(y) = \beta \int (u' \circ \sigma)(f(y - \sigma(y))z)f'(y - \sigma(y))z\phi(dz)$$ (6.50)

over interior consumption policies $\sigma$, one solution of which is the optimal policy $c^*$

Our aim is to solve the functional equation (6.50) and hence obtain $c^*$
The Coleman Operator

Recall the Bellman operator

\[ T \psi(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y - c)z) \phi(dz) \right\} \] (6.51)

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

\[ u'(c) = \beta \int (u' \circ \sigma)(f(y - c)z)f'(y - c)z\phi(dz) \] (6.52)

We call this operator the **Coleman operator** to acknowledge the work of [Col90] (although many people have studied this and other closely related iterative techniques).

In essence, \( K\sigma \) is the consumption policy that the Euler equation tells you to choose today when your future consumption policy is \( \sigma \).

The important thing to note about \( K \) is that, by construction, its fixed points coincide with solutions to the functional equation (6.50).

In particular, the optimal policy \( c^* \) is a fixed point.

Indeed, for fixed \( y \), the value \( Kc^*(y) \) is the \( c \) that solves

\[ u'(c) = \beta \int (u' \circ c^*)(f(y - c)z)f'(y - c)z\phi(dz) \]

In view of the Euler equation, this is exactly \( c^*(y) \).

Is the Coleman Operator Well Defined?

In particular, is there always a unique \( c \in (0, y) \) that solves (6.52)?

The answer is yes, under our assumptions.

For any \( \sigma \in \mathcal{P} \), the right side of (6.52)

- is continuous and strictly increasing in \( c \) on \( (0, y) \)
- diverges to \( +\infty \) as \( c \uparrow y \)

The left side of (6.52)

- 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 $\sigma \in \mathcal{P}$

### 6.9.3 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

Well prove this in just a moment

The other side to the story is the speed of the numerical implementation

It turns out that, once we actually implement these two routines, time iteration is faster and more accurate than value function iteration

More on this below

#### Equivalent Dynamics

Lets 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 $g: X \to X$ 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

$$x_{t+1} = g(x_t), \quad x_0 \text{ given}$$

Equivalently, $x_t = g^t(x_0)$, where $g$ is the $t$-th composition of $g$ with itself

Heres 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 $h: Y \to Y$ where $Y$ is another set

Suppose further that
• there exists a bijection $\tau$ from $X$ to $Y$

• the two functions commute under $\tau$, 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 $\tau^{-1}$ to both sides

$$g = \tau^{-1} \circ h \circ \tau$$

(6.53)

Here's a commutative diagram that illustrates

\[
\begin{array}{ccc}
X & \xrightarrow{g} & X \\
\downarrow{\tau} & & \uparrow{\tau^{-1}} \\
Y & \xrightarrow{h} & Y
\end{array}
\]

Here's a similar figure that traces out the action of the maps on a point $x \in X$

\[
\begin{array}{ccc}
x & \xrightarrow{g} & g(x) \\
\downarrow{\tau} & & \uparrow{\tau^{-1}} \\
\tau(x) & \xrightarrow{h} & h(\tau(x))
\end{array}
\]

Now, it's easy to check from (6.53) 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

6.9. Optimal Growth II: Time Iteration
• iterate \( n \) times with \( g \), starting at \( x \)
• shift \( x \) to \( Y \) using \( \tau \), iterate \( n \) times with \( h \) starting at \( \tau(x) \), and shift the result \( h^n(\tau(x)) \) back to \( X \) using \( \tau^{-1} \)

We end up with exactly the same object

**Back to Economics**

Have you guessed where this is leading?

What were 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, well 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

\[
Mv := h \circ v' \quad \text{where} \quad h := (u')^{-1}
\]

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

\[
M \circ T = K \circ M
\]

(6.54)

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!
6.9.4 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 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.

The Operator

Here’s some code that implements the Coleman operator:

```python
import numpy as np
from scipy.optimize import brentq

def coleman_operator(g, grid, β, u_prime, f, f_prime, shocks, Kg=None):
    ""
    The approximate Coleman operator, which takes an existing guess $g$ of the optimal consumption policy and computes and returns the updated function $Kg$ on the grid points. An array to store the new set of values $Kg$ is optionally supplied (to avoid having to allocate new arrays at each iteration). If supplied, any existing data in $Kg$ will be overwritten.
    ""
    Parameters
    --------
    g : array_like(float, ndim=1)  
The value of the input policy function on grid points
    grid : array_like(float, ndim=1)  
The set of grid points
    β : scalar  
The discount factor
    u_prime : function  
The derivative $u'(c)$ of the utility function
    f : function  
The production function $f(k)$
    f_prime : function  
The derivative $f'(k)$
    shocks : numpy array  
An array of draws from the shock, for Monte Carlo integration (to compute expectations).
    Kg : array_like(float, ndim=1) optional (default=None)  
Array to write output values to
    ""
    # === Apply linear interpolation to g === #
```

6.9. Optimal Growth II: Time Iteration
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:

```python
from scipy.optimize import fminbound

def bellman_operator(w, grid, β, u, f, shocks, Tw=None, compute_policy=0):
    
    The approximate Bellman operator, which computes and returns the updated value function \( T_w \) on the grid points. An array to store the new set of values \( T_w \) is optionally supplied (to avoid having to allocate new arrays at each iteration). If supplied, any existing data in \( T_w \) will be overwritten.

    Parameters
    ----------
    w : array_like(float, ndim=1)
        The value of the input function on different grid points
    grid : array_like(float, ndim=1)
        The set of grid points
    β : scalar
        The discount factor
    u : function
        The utility function
    f : function
        The production function
    shocks : numpy array
        An array of draws from the shock, for Monte Carlo integration (to compute expectations).
    Tw : array_like(float, ndim=1) optional (default=None)
        Array to write output values to
```
compute_policy : Boolean, optional (default=False)
    Whether or not to compute policy function

# === Apply linear interpolation to w === #
w_func = lambda x: np.interp(x, grid, w)

# == Initialize Tw if necessary == #
if Tw is None:
    Tw = np.empty_like(w)

if compute_policy:
    σ = np.empty_like(w)

# == set Tw[i] = max_c { u(c) + β E w(f(y - c) z)} == #
for i, y in enumerate(grid):
    def objective(c):
        return -u(c) - β * np.mean(w_func(f(y - c) * shocks))
    c_star = fminbound(objective, 1e-10, y)
    if compute_policy:
        σ[i] = c_star
    Tw[i] = - objective(c_star)

if compute_policy:
    return Tw, σ
else:
    return Tw

Testing on the Log / Cobb–Douglas case

As we *did for value function iteration*, lets start by testing our method in the presence of a model that does have an analytical solution.

We assume the following imports

```python
import matplotlib.pyplot as plt
import quantecon as qe
```

Now lets bring in the log-linear growth model we used in the *value function iteration lecture*

```python
class LogLinearOG:
    """
    Log linear optimal growth model, with log utility, CD production and
    multiplicative lognormal shock, so that

    \[ y = f(k, z) = z k^α \]

    with \( z \sim LN(μ, s) \).

    The class holds parameters and true value and policy functions.
    """
```

6.9. Optimal Growth II: Time Iteration
```python

# Initialize

def __init__(self, α=0.4, β=0.96, μ=0, s=0.1):
    self.α, self.β, self.μ, self.s = α, β, μ, s
    
    # Some useful constants
    self.ab = α * β
    self.c1 = np.log(1 - self.ab) / (1 - β)
    self.c2 = (μ + α * np.log(self.ab)) / (1 - α)
    self.c3 = 1 / (1 - β)
    self.c4 = 1 / (1 - self.ab)

def u(self, c):
    "Utility"
    return np.log(c)

def u_prime(self, c):
    return 1 / c

def f(self, k):
    "Deterministic part of production function."
    return k ** self.α

def f_prime(self, k):
    return self.α * k ** (self.α - 1)

def c_star(self, y):
    "True optimal policy."
    return (1 - self.α * self.β) * y

def v_star(self, y):
    "True value function."
    return self.c1 + self.c2 * (self.c3 - self.c4) + self.c4 * np.log(y)

# Unpack parameters / functions for convenience

lg = LogLinearOG()

alpha, beta, mu, s = lg.α, lg.β, lg.μ, lg.s
v_star, c_star = lg.v_star, lg.c_star
u, u_prime, f, f_prime = lg.u, lg.u_prime, lg.f, lg.f_prime

# We also need a grid and some shock draws for Monte Carlo integration

grid_max = 4  # Largest grid point
grid_size = 200  # Number of grid points
shock_size = 250  # Number of shock draws in Monte Carlo integral

grid = np.linspace(1e-5, grid_max, grid_size)
shocks = np.exp(μ + s * np.random.randn(shock_size))
```

Next we generate an instance

```python
lg = LogLinearOG()

alpha, beta, mu, s = lg.α, lg.β, lg.μ, lg.s
v_star, c_star = lg.v_star, lg.c_star
u, u_prime, f, f_prime = lg.u, lg.u_prime, lg.f, lg.f_prime
```

We also need a grid and some shock draws for Monte Carlo integration

```python
grid_max = 4  # Largest grid point
grid_size = 200  # Number of grid points
shock_size = 250  # Number of shock draws in Monte Carlo integral

grid = np.linspace(1e-5, grid_max, grid_size)
shocks = np.exp(μ + s * np.random.randn(shock_size))
```
As a preliminary test, let’s see if $Kc^* = c^*$, as implied by the theory

```python
c_star_new = coleman_operator(c_star(grid),
                          grid, β, u_prime,
                          f, f_prime, shocks)
```

`fig, ax = plt.subplots()
ax.plot(grid, c_star(grid), label="optimal policy $c^*$")
ax.plot(grid, c_star_new, label="$Kc^*$")
ax.legend(loc='upper left')
plt.show()`

Here’s the result:

![Graph showing optimal policies](image)

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 $c^*$.

The initial condition we will use is the one that eats the whole pie: $c(y) = y$
We see that the policy has converged nicely, in only a few steps.
1. $K^n c$ where $c(y) = y$

2. $(M \circ T^n \circ M^{-1}) c$ where $c(y) = y$

In each case we will compare the resulting policy to $c^*$

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.

```python
# Parameters
alpha, beta, mu, s = lg.alpha, lg.beta, lg.mu, lg.s
v_star, c_star = lg.v_star, lg.c_star
u, u_prime, f, f_prime = lg.u, lg.u_prime, lg.f, lg.f_prime

# Initial conditions
g_init = grid
w_init = u(grid)
sim_length = 20

g, w = g_init, w_init
for i in range(sim_length):
    new_g = coleman_operator(g, grid, beta, u_prime, f, f_prime, shocks)
    new_w = bellman_operator(w, grid, u, f, shocks)
    g, w = new_g, new_w

new_w, vf_g = bellman_operator(w, grid, u, f, shocks, compute_policy=True)

fig, ax = plt.subplots()

pf_error = c_star(grid) - g
vf_error = c_star(grid) - vf_g

ax.plot(grid, 0 * grid, 'k-', lw=1)
ax.plot(grid, pf_error, lw=2, alpha=0.6, label="policy iteration error")
ax.plot(grid, vf_error, lw=2, alpha=0.6, label="value iteration error")

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

Here is the result, which shows the errors in each case.
As you can see, time iteration is much more accurate for a given number of iterations.

### 6.9.5 Exercises

#### Exercise 1

Show that (6.54) is valid. In particular,

- Let \( v \) be strictly concave and continuously differentiable on \((0, \infty)\)
- Fix \( y \in (0, \infty) \) and show that \( MTv(y) = KMv(y) \)

#### Exercise 2

Show that \( M \) is a bijection from \( \mathcal{V} \) to \( \mathcal{P} \)
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 \( c(y) = y \)
- start value function iteration from \( v(y) = u(y) \)
- set \( \gamma = 1.5 \)

Compare the resulting policies and check that they are close

Exercise 4

Do the same exercise, but now, rather than plotting results, time how long 20 iterations takes in each case

6.9.6 Solutions

Solution to Exercise 1

Let \( T, K, M, v \) and \( y \) be as stated in the exercise

Using the envelope theorem, one can show that \( (Tv)'(y) = u'(c(y)) \) where \( c(y) \) solves

\[ u'(c(y)) = \beta \int v'(f(y - c(y))z)f'(y - c(y))z\phi(dz) \] (6.55)

Hence \( MTv(y) = (u')^{-1}(u'(c(y))) = c(y) \)

On the other hand, \( KMv(y) \) is the \( c(y) \) that solves

\[ u'(c(y)) = \beta \int (u' \circ (Mv))(f(y - c(y))z)f'(y - c(y))z\phi(dz) \]

\[ = \beta \int (u' \circ ((u')^{-1} \circ v'))(f(y - c(y))z)f'(y - c(y))z\phi(dz) \]

\[ = \beta \int v'(f(y - c(y))z)f'(y - c(y))z\phi(dz) \]

We see that \( c(y) \) is the same in each case
Solution to 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 $\mathbb{R}_+$.

Solution to Exercise 3

Here's the code, which will execute if you've run all the code above.

```python
## Define the model
alpha = 0.65
beta = 0.95
mu = 0
s = 0.1
grid_min = 1e-6
grid_max = 4
grid_size = 200
shock_size = 250

gamma = 1.5  # Preference parameter

def f(k):
    return k**alpha

def f_prime(k):
    return alpha * k**(alpha - 1)
```

Chapter 6. Dynamic Programming
def u(c):
    return (c**(1 - γ) - 1) / (1 - γ)

def u_prime(c):
    return c**(-γ)

grid = np.linspace(grid_min, grid_max, grid_size)
shocks = np.exp(μ + s * np.random.randn(shock_size))

## Let's make convenience functions based around these primitives

def crra_bellman(w):
    return bellman_operator(w, grid, β, u, f, shocks)

def crra_coleman(g):
    return coleman_operator(g, grid, β, u_prime, f, f_prime, shocks)

## Iterate with K and T, compare policies

g_init = grid
w_init = u(grid)
sim_length = 20

g, w = g_init, w_init
for i in range(sim_length):
    new_g = crra_coleman(g)
    new_w = crra_bellman(w)
    g, w = new_g, new_w
    new_w, vf_g = bellman_operator(w, grid, β, u, f, shocks, compute_policy=True)

fig, ax = plt.subplots()

ax.plot(grid, g, lw=2, alpha=0.6, label="policy iteration")
ax.plot(grid, vf_g, lw=2, alpha=0.6, label="value iteration")
ax.legend(loc="upper left")
plt.show()

Heres the resulting figure
The policies are indeed close

**Solution to Exercise 4**

Here is the code

It assumes that you've just run the code from the previous exercise

```python
g_init = grid
w_init = u(grid)
sim_length = 100

print("Timing value function iteration")

w = w_init
qe.util.tic()
for i in range(sim_length):
    new_w = crra_bellman(w)
    w = new_w
qe.util.toc()
```
print("Timing Euler equation time iteration")

g = g_init
g_methods = "tic()
for i in range(sim_length):  
    new_g = crra_coleman(g)
    g = new_g
qe.util.toc()

If you run this youll find that the two operators execute at about the same speed
However, as we saw above, time iteration is numerically far more accurate for a given number of iterations

6.10 Optimal Growth III: The Endogenous Grid Method

Contents

• Optimal Growth III: The Endogenous Grid Method
  – Overview
  – Key Idea
  – Implementation
  – Speed

6.10.1 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 well 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 [Car06]
6.10.2 Key Idea

Let's start by reminding ourselves of the theory and then see how the numerics fit in.

Theory

Take the model set out in the time iteration lecture, following the same terminology and notation.

The Euler equation is

\[(u' \circ c^*)(y) = \beta \int (u' \circ c^*)(f(y - c^*(y))z)f'(y - c^*(y))z \phi(dz)\]  

(6.56)

As we saw, the Coleman operator is a nonlinear operator \(K\) engineered so that \(c^*\) is a fixed point of \(K\).

It takes as its argument a continuous strictly increasing consumption policy \(g \in \Sigma\).

It returns a new function \(Kg\), where \((Kg)(y)\) is the \(c \in (0, \infty)\) that solves

\[u'(c) = \beta \int (u' \circ g)(f(y - c)z)f'(y - c)z \phi(dz)\]  

(6.57)

Exogenous Grid

As discussed in the lecture on time iteration, to implement the method on a computer we need 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 (6.57) and a root finding routine

Each \(c_i\) is then interpreted as the value of the function \(Kg\) at \(y_i\).

Thus, with the points \(\{y_i, c_i\}\) in hand, we can reconstruct \(Kg\) via approximation.

Iteration then continues.

Endogenous Grid

The method discussed above requires a root finding routine to find the \(c_i\) corresponding to a given income value \(y_i\).

Root finding is costly because it typically involves a significant number of function evaluations.
As pointed out by Carroll [Car06], we can avoid this if \( y_i \) is chosen endogenously

The only assumption required is that \( u' \) is invertible on \((0, \infty)\)

The idea is this:
First we fix an exogenous grid \( \{k_i\} \) for capital \((k = y - c)\)
Then we obtain \( c_i \) via

\[
c_i = (u')^{-1} \left\{ \beta \int (u' \circ g)(f(k_i)z) f'(k_i) z \phi(dz) \right\}
\]

(6.58)

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 (6.57)

With the points \(\{y_i, c_i\}\) in hand, we can reconstruct \(Kg\) via approximation as before
The name EGM comes from the fact that the grid \(\{y_i\}\) is determined endogenously

### 6.10.3 Implementation

Lets implement this version of the Coleman operator and see how it performs

The Operator

Heres an implementation of \(K\) using EGM as described above

```python
import numpy as np

def coleman_egm(g, k_grid, \(\beta\), u_prime, u_prime_inv, \(f\), f_prime, shocks):
    
    """
    The approximate Coleman operator, updated using the endogenous grid
    method.
    """

    Parameters
    """
    g : function
        The current guess of the policy function
    k_grid : array_like(float, ndim=1)
        The set of *exogenous* grid points, for capital \(k = y - c\)
    \(\beta\) : scalar
        The discount factor
    u_prime : function
        The derivative \(u'(c)\) of the utility function
    u_prime_inv : function
        The inverse of \(u'\) (which exists by assumption)
    \(f\) : function
        The production function \(f(k)\)
    """
```
f_prime : function
    The derivative f'(k)
shocks : numpy array
    An array of draws from the shock, for Monte Carlo integration (to
    compute expectations).

""

# Allocate memory for value of consumption on endogenous grid points
c = np.empty_like(k_grid)

# Solve for updated consumption value
for i, k in enumerate(k_grid):
    vals = u_prime(g(f(k) * shocks)) * f_prime(k) * shocks
    c[i] = u_prime_inv(\beta * np.mean(vals))

# Determine endogenous grid
y = k_grid + c   # y_i = k_i + c_i

# Update policy function and return
Kg = lambda x: np.interp(x, y, c)
return Kg

Note the lack of any root finding algorithm

Well also run our original implementation, which uses an exogenous grid and requires root finding, so we
can perform some comparisons

import numpy as np
from scipy.optimize import brentq

def coleman_operator(g, grid, \beta, u_prime, f, f_prime, shocks, Kg=None):
    ""
    The approximate Coleman operator, which takes an existing guess g of the
    optimal consumption policy and computes and returns the updated function
    Kg on the grid points. An array to store the new set of values Kg is
    optionally supplied (to avoid having to allocate new arrays at each
    iteration). If supplied, any existing data in Kg will be overwritten.

    Parameters
    --------
    g : array_like(float, ndim=1)
        The value of the input policy function on grid points
    grid : array_like(float, ndim=1)
        The set of grid points
    \beta : scalar
        The discount factor
    u_prime : function
        The derivative u'(c) of the utility function
    f : function
        The production function f(k)
    f_prime : function
        The derivative f'(k)
    ""
shocks : numpy array
An array of draws from the shock, for Monte Carlo integration (to
compute expectations).
Kg : array_like(float, ndim=1) optional (default=None)
Array to write output values to

"""# === Apply linear interpolation to g === #
g_func = lambda x: np.interp(x, grid, g)
# == Initialize Kg if necessary == #
if Kg is None:
    Kg = np.empty_like(g)
# == solve for updated consumption value
for i, y in enumerate(grid):
    def h(c):
        vals = u_prime(g_func(f(y - c) * shocks)) * f_prime(y - c) * _
        return u_prime(c) - np.mean(vals)
    c_star = brentq(h, 1e-10, y - 1e-10)
    Kg[i] = c_star

return Kg

Let's test out the code above on some example parameterizations, after the following imports

import matplotlib.pyplot as plt
import quantecon as qe

Testing on the Log / Cobb–Douglas case

As we did for value function iteration and time iteration, lets start by testing our method with the log-linear benchmark
The first step is to bring in the log-linear growth model that we used in the value function iteration lecture

class LogLinearOG:
    ""
    Log linear optimal growth model, with log utility, CD production and
    multiplicative lognormal shock, so that
    
    \[ y = f(k, z) = z k^{\alpha} \]
    
    with \( z \sim LN(\mu, s) \).
    The class holds parameters and true value and policy functions.
    """
```python
def __init__(self, α=0.4, β=0.96, μ=0, s=0.1):

    self.α, self.β, self.μ, self.s = α, β, μ, s

    # == Some useful constants == #
    self.ab = α * β
    self.c1 = np.log(1 - self.ab) / (1 - β)
    self.c2 = (μ + α * np.log(self.ab)) / (1 - α)
    self.c3 = 1 / (1 - β)
    self.c4 = 1 / (1 - self.ab)

def u(self, c):
    "Utility"
    return np.log(c)

def u_prime(self, c):
    return 1 / c

def f(self, k):
    "Deterministic part of production function."
    return k**self.α

def f_prime(self, k):
    return self.α * k**(self.α - 1)

def c_star(self, y):
    "True optimal policy."
    return (1 - self.α * self.β) * y

def v_star(self, y):
    "True value function."
    return self.c1 + self.c2 + (self.c3 - self.c4) + self.c4 * np.log(y)

Next we generate an instance

lg = LogLinearOG()

# == Unpack parameters / functions for convenience == #
α, β, μ, s = lg.α, lg.β, lg.μ, lg.s
v_star, c_star = lg.v_star, lg.c_star
u, u_prime, f, f_prime = lg.u, lg.u_prime, lg.f, lg.f_prime

We also need a grid over capital and some shock draws for Monte Carlo integration

grid_max = 4 # Largest grid point, exogenous grid
grid_size = 200 # Number of grid points
shock_size = 250 # Number of shock draws in Monte Carlo integral

k_grid = np.linspace(1e-5, grid_max, grid_size)
shocks = np.exp(μ + s * np.random.randn(shock_size))

As a preliminary test, lets see if $Kc^* = c^*$, as implied by the theory
```

626 Chapter 6. Dynamic Programming
c_star_new = coleman_egm(c_star,
    k_grid, β, u_prime, u_prime, f, f_prime, shocks)

fig, ax = plt.subplots(figsize=(9, 6))

ax.plot(k_grid, c_star(k_grid), label="optimal policy $c^*$")
ax.plot(k_grid, c_star_new(k_grid), label="$Kc^*$")

ax.legend(loc='upper left')
plt.show()

Notice that were passing $u'_\prime$ to coleman_egm twice.
The reason is that, in the case of log utility, $u'(c) = (u')^{-1}(c) = 1/c$
Hence $u'_\prime$ and $u'_\prime^{-1}$ are the same.
In any case, heres the result:

We cant really distinguish the two plots.
In fact its easy to see that the difference is essentially zero:

\[
\max(abs(c_{\text{star\_new}}(k_{\text{grid}}) - c_{\text{star}}(k_{\text{grid}))))
\]
Next let's try iterating from an arbitrary initial condition and see if we converge towards $c^*$

Let's start from the consumption policy that eats the whole pie: $c(y) = y$

```python
g = lambda x: x
n = 15
fig, ax = plt.subplots(figsize=(9, 6))
lb = 'initial condition $c(y) = y$'
ax.plot(k_grid, g(k_grid), color=plt.cm.jet(0), lw=2, alpha=0.6, label=lb)
for i in range(n):
    new_g = coleman_egm(g, k_grid, β, u_prime, u_prime, f, f_prime, shocks)
    g = new_g
    ax.plot(k_grid, g(k_grid), color=plt.cm.jet(i / n), lw=2, alpha=0.6)
lb = 'true policy function $c^*$'
ax.plot(k_grid, c_star(k_grid), 'k-', lw=2, alpha=0.8, label=lb)
ax.legend(loc='upper left')
plt.show()
```

We see that the policy has converged nicely, in only a few steps.
6.10.4 Speed

Now lets compare the clock times per iteration for the standard Coleman operator (with exogenous grid) and the EGM version

Well do so using the CRRA model adopted in the exercises of the *Euler equation time iteration lecture*

Heres the model and some convenient functions

```python
## Define the model
\alpha = 0.65
\beta = 0.95
\mu = 0
s = 0.1
grid_min = 1e-6
grid_max = 4
grid_size = 200
shock_size = 250

\gamma = 1.5  # Preference parameter
\gamma_inv = 1 / \gamma

def f(k):
    return k**\alpha

def f_prime(k):
    return \alpha * k**(\alpha - 1)

def u(c):
    return (c**(1 - \gamma) - 1) / (1 - \gamma)

def u_prime(c):
    return c**(-\gamma)

def u_prime_inv(c):
    return c**(-\gamma_inv)

k_grid = np.linspace(grid_min, grid_max, grid_size)
shocks = np.exp(s * np.random.randn(shock_size))

## Let's make convenience functions based around these primitives

def crra_coleman(g):
    return coleman_operator(g, k_grid, \beta, u_prime, f, f_prime, shocks)

def crra_coleman_egm(g):
    return coleman_egm(g, k_grid, \beta, u_prime, u_prime_inv, f, f_prime, shocks)
```

Heres the result

```python
## Iterate, compare policies

sim_length = 20
```
```
print("Timing standard Coleman policy function iteration")
g_init = k_grid
g = g_init
qe.util.tic()
for i in range(sim_length):
    new_g = crra_coleman(g)
    g = new_g
qe.util.toc()

print("Timing policy function iteration with endogenous grid")
g_init_egm = lambda x: x
g = g_init_egm
qe.util.tic()
for i in range(sim_length):
    new_g = crra_coleman_egm(g)
    g = new_g
qe.util.toc()
```

Timing standard Coleman policy function iteration
TOC: Elapsed: 2.307983636856079 seconds.
Timing policy function iteration with endogenous grid
TOC: Elapsed: 0.34932971000671387 seconds.

We see that the EGM version is more than 6 times faster.

At the same time, the absence of numerical root finding means that it is typically more accurate at each step as well.

### 6.11 LQ Dynamic Programming Problems
6.11.1 Overview

Linear quadratic (LQ) control refers to a class of dynamic optimization problems that have found applications in almost every scientific field.

This lecture provides an introduction to LQ control and its economic applications.

As we will see, LQ systems have a simple structure that makes them an excellent workhorse for a wide variety of economic problems.

Moreover, while the linear-quadratic structure is restrictive, it is in fact far more flexible than it may appear initially.

These themes appear repeatedly below.

Mathematically, LQ control problems are closely related to the Kalman filter.

- Recursive formulations of linear-quadratic control problems and Kalman filtering problems both involve matrix Riccati equations.
- Classical formulations of linear control and linear filtering problems make use of similar matrix decompositions (see for example this lecture and this lecture).

In reading what follows, it will be useful to have some familiarity with:

- matrix manipulations
- vectors of random variables
- dynamic programming and the Bellman equation (see for example this lecture and this lecture)

For additional reading on LQ control, see, for example,

- [LS18], chapter 5
- [HS08], chapter 4
- [HLL96], section 3.5

In order to focus on computation, we leave longer proofs to these sources (while trying to provide as much intuition as possible).

6.11.2 Introduction

The linear part of LQ is a linear law of motion for the state, while the quadratic part refers to preferences.

Let’s begin with the former, move on to the latter, and then put them together into an optimization problem.

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
\[ x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \quad t = 0, 1, 2, \ldots \quad (6.59) \]

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 \( \mathbb{E}w_tw'_t = I \), 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 \( (6.59) \), 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 \( \mu \)

Another alteration that is useful to introduce (well 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
\[ a_{t+1} = (1 + r)a_t - u_t - \bar{c} + \sigma w_{t+1} + \mu \]  

(6.60)

How can we write this new system in the form of equation (6.59)?

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

\[
\begin{pmatrix}
a_{t+1} \\
1
\end{pmatrix} = \begin{pmatrix}
1 + r & -\bar{c} + \mu \\
0 & 1
\end{pmatrix} \begin{pmatrix}
a_t \\
1
\end{pmatrix} + \begin{pmatrix}
-1 \\
0
\end{pmatrix} u_t + \begin{pmatrix}
\sigma \\
0
\end{pmatrix} w_{t+1}
\]

(6.61)

then the first row is equivalent to (6.60).

Moreover, the model is now linear, and can be written in the form of (6.59) by setting

\[
x_t := \begin{pmatrix}
a_t \\
1
\end{pmatrix}, \quad A := \begin{pmatrix}
1 + r & -\bar{c} + \mu \\
0 & 1
\end{pmatrix}, \quad B := \begin{pmatrix}
-1 \\
0
\end{pmatrix}, \quad C := \begin{pmatrix}
\sigma \\
0
\end{pmatrix}
\]

(6.62)

In effect, we’ve bought ourselves linearity by adding another state.

**Preferences**

In the LQ model, the aim is to minimize a flow of losses, where time-\( t \) loss is given by the quadratic expression

\[ x_t' R x_t + u_t' Q u_t \]  

(6.63)

Here

- \( R \) is assumed to be \( n \times n \), symmetric and nonnegative definite
- \( Q \) is assumed to be \( k \times k \), symmetric and positive definite

**Note:** In fact, for many economic problems, the definiteness conditions on \( R \) and \( Q \) can be relaxed. It is sufficient that certain submatrices of \( R \) and \( Q \) be nonnegative definite. See [HS08] for details.
Example 1

A very simple example that satisfies these assumptions is to take $R$ and $Q$ to be identity matrices, so that current loss is

$$x_t' I x_t + u_t' I u_t = \|x_t\|^2 + \|u_t\|^2$$

Thus, for both the state and the control, loss is measured as squared distance from the origin.

(Indeed the general case (6.63) 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

$$x_t' Rx_t + u_t' Qu_t = u_t^2 = (c_t - \bar{c})^2$$

Under this specification, the household’s current loss is the squared deviation of consumption from the ideal level $\bar{c}$.

6.11.3 Optimality – Finite Horizon

Let us now be precise about the optimization problem we wish to consider, and look at how to solve it.

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 $\{u_0, \ldots, u_{T-1}\}$ to minimize the objective

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (x_t' Rx_t + u_t' Qu_t) + \beta^T x_T' R_f x_T \right\}$$

subject to the law of motion (6.59) and initial state $x_0$.

The new objects introduced here are $\beta$ and the matrix $R_f$.

The scalar $\beta$ 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 $w_1, \ldots, w_T$

**Information**

There is 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 $\{u_0, \ldots, u_{T-1}\}$.

When choosing these controls, the decision maker is permitted to take into account the effects of the shocks $\{w_1, \ldots, w_T\}$ 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 $\sigma$-algebra generated by $x_0, w_1, w_2, \ldots, w_t$.

This is in fact equivalent to stating that $u_t$ can be written in the form $u_t = g_t(x_0, w_1, w_2, \ldots, w_t)$ 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 $x_0, w_1, w_2, \ldots, w_t$.

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 $x_t$.

**Solution**

To solve the finite horizon LQ problem we can use a dynamic programming strategy based on backwards 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

$$\min_u \{x_{T-1}'Rx_{T-1} + u'Qu + \beta \mathbb{E}J_T(Ax_{T-1} + Bu + Cw_T)\}$$

At this stage, it is convenient to define the function

$$J_{T-1}(x) = \min_u \{x'Rx + u'Qu + \beta \mathbb{E}J_T(Ax + Bu + Cw_T)\} \quad (6.65)$$

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 $J_T(x) = x'Rf$ 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 $x_{T-1} = Ax_{T-2} + Bu + Cw_{T-1}$, 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

$$\min_u \{x_{T-2}'Rx_{T-2} + u'Qu + \beta \mathbb{E}J_{T-1}(Ax_{T-2} + Bu + Cw_{T-1})\}$$

Letting

$$J_{T-2}(x) = \min_u \{x'Rx + u'Qu + \beta \mathbb{E}J_{T-1}(Ax + Bu + Cw_{T-1})\}$$

the pattern for backwards induction is now clear.

In particular, we define a sequence of value functions $\{J_0, \ldots, J_T\}$ via

$$J_{t-1}(x) = \min_u \{x'Rx + u'Qu + \beta \mathbb{E}J_t(Ax + Bu + Cw_t)\} \quad \text{and} \quad J_T(x) = x'Rf$$

The first equality is the Bellman equation from dynamic programming theory specialized to the finite horizon LQ problem.

Now that we have $\{J_0, \ldots, J_T\}$, we can obtain the optimal controls.

As a first step, let's find out what the value functions look like.

It turns out that every $J_t$ has the form $J_t(x) = x'P_tx + d_t$ 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, (6.65) becomes

$$J_{T-1}(x) = \min_u \{x'Rx + u'Qu + \beta \mathbb{E}(Ax + Bu + Cw_T)'P_T(Ax + Bu + Cw_T)\} \quad (6.66)$$
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

\[
u = -(Q + \beta B'P_T B)^{-1}\beta B'P_T Ax
\]  

(6.67)

Plugging this back into (6.66) and rearranging yields

\[J_{T-1}(x) = x'P_{T-1}x + d_{T-1}\]

where

\[P_{T-1} = R - \beta^2 A'P_T B(Q + \beta B'P_T B)^{-1}B'P_T A + \beta A'P_T A\]  

(6.68)

and

\[d_{T-1} := \beta \text{trace}(C'P_T C)\]  

(6.69)

(The algebra is a good exercise; well 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

\[P_{t-1} = R - \beta^2 A'P_t B(Q + \beta B'P_t B)^{-1}B'P_t A + \beta A'P_t A \quad \text{with} \quad P_T = R_f\]  

(6.70)

and

\[d_{t-1} = \beta(d_t + \text{trace}(C'P_t C)) \quad \text{with} \quad d_T = 0\]  

(6.71)

Recalling (6.67), the minimizers from these backward steps are

\[u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B'P_{t+1} B)^{-1}\beta B'P_{t+1} A\]  

(6.72)

These are the linear optimal control policies we *discussed above*.

In particular, the sequence of controls given by (6.72) and (6.59) solves our finite horizon LQ problem.

Rephrasing this more precisely, the sequence \( u_0, \ldots, u_{T-1} \) given by

\[u_t = -F_t x_t \quad \text{with} \quad x_{t+1} = (A - BF_t)x_t + Cw_{t+1}\]  

(6.73)

for \( t = 0, \ldots, T - 1 \) attains the minimum of (6.64) subject to our constraints.
6.11.4 Implementation

We will use code from lqcontrol.py in QuantEcon.py to solve finite and infinite horizon linear quadratic control problems.

In the module, the various updating, simulation and fixed point methods are wrapped in a class called LQ, which includes:

- **Instance data:**
  - The required parameters $Q, R, A, B$ and optional parameters $C, \beta, T, R_f, N$ specifying a given LQ model
    - * set $T$ and $R_f$ to None in the infinite horizon case
    - * set $C = \text{None}$ (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 (6.70), (6.71) and (6.72)
  - **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

### 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 consumers preference for an intertemporally smooth consumption stream.

(See, for example, [Fri56] or [MB54])

One property of those models is that households purchase and sell financial assets to make consumption streams smoother than income streams.

The household savings problem outlined above captures these ideas.

The optimization problem for the household is to choose a consumption sequence in order to minimize

$$
\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (c_t - \bar{c})^2 + \beta^T qa_T^2 \right\}
$$

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 (6.60)

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 (6.62)

To match with this state and control, the objective function (6.74) can be written in the form of (6.64) 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 (6.70) and (6.72)

After generating shocks $w_1, \ldots, w_T$, the dynamics for assets and consumption can be simulated via (6.73)

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

```python
import numpy as np
import matplotlib.pyplot as plt
from quantecon import LQ

# == Model parameters == #
r = 0.05
beta = 1/(1 + r)
T = 45
c_bar = 2
sigma = 0.25
mu = 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 + mu],
     [0, 1]]
B = [[-1],
     [0]]
C = [[sigma],
     [0]]

# == Compute solutions and simulate == #
lq = LQ(Q, R, A, B, C, beta=beta, T=T, Rf=Rf)
x0 = (0, 1)
xp, up, wp = lq.compute_sequence(x0)
```

6.11. LQ Dynamic Programming Problems
# == Convert back to assets, consumption and income == #
assets = xp[:, :]
# a_t

c = up.flatten() + c_bar   # c_t
income = σ * wp[:, 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, lets increase $\beta$ 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

```python
# == 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
income = sigma * wp[0, 1:] + mu  # y_t

c = up.flatten() + c_bar  # c_t

def plot_results(n_rows=2)
    fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))
    plt.subplots_adjust(hspace=0.5)
    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 - mu), '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.

### 6.11.5 Extensions and Comments

Let's now consider a number of standard extensions to the LQ problem treated above.

#### Time-Varying Parameters

In some settings it can be desirable to allow $A, B, C, R$ and $Q$ to depend on $t$.

For the sake of simplicity, we've chosen not to treat this extension in our implementation given below.
However, the loss of generality is not as large as you might first imagine. In fact, we can tackle many models with time-varying parameters by suitable choice of state variables. One illustration is given below.

For further examples and a more systematic treatment, see [HS13], section 2.4

**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

$$
E \left\{ \sum_{t=0}^{T-1} \beta^t (x_t' R x_t + u_t' Q u_t + 2 u_t' N x_t) + \beta^T x_T' R_f x_T \right\} \quad (6.75)
$$

Our results extend to this case in a straightforward way.

The sequence $\{P_t\}$ from (6.70) becomes

$$
P_{t-1} = R - (\beta B' P_t A + N)' (Q + \beta B' P_t B)^{-1} (\beta B' P_t A + N) + \beta A' P_t A \quad \text{with} \quad P_T = R_f \quad (6.76)
$$

The policies in (6.72) are modified to

$$
u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B' P_{t+1} B)^{-1} (\beta B' P_{t+1} A + N) \quad (6.77)
$$

The sequence $\{d_t\}$ is unchanged from (6.71).

We leave interested readers to confirm these results (the calculations are long but not overly difficult).

**Infinite Horizon**

Finally, we consider the infinite horizon case, with cross-product term, unchanged dynamics and objective function given by

$$
E \left\{ \sum_{t=0}^{\infty} \beta^t (x_t' R x_t + u_t' Q u_t + 2 u_t' N x_t) \right\} \quad (6.78)
$$

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

\[ P = R - (\beta B'PA + N)'(Q + \beta B'PB)^{-1}(\beta B'PA + N) + \beta A'PA \]  

(6.79)

Equation (6.79) is also called the LQ Bellman equation, and the map that sends a given $P$ into the right-hand side of (6.79) is called the LQ Bellman operator.

The stationary optimal policy for this model is

\[ u = -Fx \quad \text{where} \quad F = (Q + \beta B'PB)^{-1}(\beta B'PA + N) \]  

(6.80)

The sequence $\{d_t\}$ from (6.71) is replaced by the constant value

\[ d := \text{trace}(C'PC)^{-\frac{\beta}{1-\beta}} \]  

(6.81)

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.

**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 (6.80) or (6.77).

It follows that we can ignore uncertainty when solving for optimal behavior, and plug it back in when examining optimal state dynamics.

**6.11.6 Further Applications**

**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

\[
E \left\{ \sum_{t=0}^{T-1} \beta^t (c_t - \bar{c})^2 + \beta^T q a_T^2 \right\} \tag{6.82}
\]

subject to \( a_{t+1} = (1 + r)a_t - c_t + y_t, \quad t \geq 0 \)

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

\[
a_{t+1} = (1 + r)a_t - u_t - \bar{c} + m_1 t + m_2 t^2 + \sigma w_{t+1} \tag{6.83}
\]

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

\[
x_t := \begin{pmatrix} a_t \\ 1 \\ t \\ t^2 \end{pmatrix}, \quad A := \begin{pmatrix} 1 + r & -\bar{c} & m_1 & m_2 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{6.84}
\]

If you expand the expression \( x_{t+1} = Ax_t + Bu_t + Cw_{t+1} \) using this specification, you will find that assets follow (6.83) as desired, and that the other state variables also update appropriately

To implement preference specification (6.82) we take

\[
Q := 1, \quad R := \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad R_f := \begin{pmatrix} q & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{6.85}
\]

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.

**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

$$y_t = \begin{cases} p(t) + \sigma w_{t+1} & \text{if } t \leq K \\ s & \text{otherwise} \end{cases} \quad (6.86)$$

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 (6.74)

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 (6.86) 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 consumers 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 backwards 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 backwards 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.

**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

\[
\mathbb{E}\left\{ \sum_{t=0}^{\infty} \beta^t \pi_t \right\} \quad \text{where} \quad \pi_t := p_t q_t - c q_t - \gamma(q_{t+1} - q_t)^2
\]

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 lets 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

Its 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 \( \gamma \) is that

- if \( \gamma \) is close to zero, then \( q_t \) will track the time path of \( \bar{q}_t \) relatively closely
- if \( \gamma \) 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 \( \gamma \)
dynamics with $\gamma = 1$
dynamics with $\gamma = 10$
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 \( x_t = (q_t \, q_t^1)' \), while the control is chosen as \( u_t = q_{t+1} - q_t \).

We also manipulated the profit function slightly.

In (6.87), current profits are \( \pi_t := p_t q_t - c q_t - \gamma (q_{t+1} - q_t)^2 \).

Let's now replace \( \pi_t \) in (6.87) with \( \hat{\pi}_t := \pi_t - a_1 q_t^2 \).

This makes no difference to the solution, since \( a_1 q_t^2 \) does not depend on the controls.

(In fact we are just adding a constant term to (6.87), 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

\[ \min \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left\{ a_1 (q_t - \bar{q}_t)^2 + \gamma u_t^2 \right\} \]
Its now relatively straightforward to find $R$ and $Q$ such that (6.88) can be written as (6.78).

Furthermore, the matrices $A, B$ and $C$ from (6.59) 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.

## 6.11.7 Exercises

### 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 $q = 10^4$.

### 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 (6.84):

- 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 (6.85).

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.

### 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 $\gamma$ varies between 1 and 50 (see figures).
6.11.8 Solutions

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
- \( p(T) = 0 \).

```python
# == Model parameters == #
r = 0.05
beta = 1/(1 + r)
T = 50
c_bar = 1.5
sigma = 0.15
mu = 2
q = 1e4
m1 = T * (mu/(T/2)**2)
m2 = -(mu/(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 = [[sigma],
     [0],
     [0],
     [0]]

# == Compute solutions and simulate == #
lq = LQ(Q, R, A, B, C, beta=beta, 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
p = up.flatten() + c_bar  # Consumption
T = np.arange(1, T+1)
income = \sigma * wp[0, 1:] + m1 * time + m2 * time**2  # Income

# == Plot results == #
fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))
plt.subplots_adjust(hspace=0.5)
bbox = (0., 1.02, 1., .102)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.7}

axes[0].plot(range(1, T+1), income, 'g-', label="non-financial income", **p_args)
axes[0].plot(range(T), c, 'k-', label="consumption", **p_args)
axes[1].plot(range(T+1), ap.flatten(), 'b-', label="assets", **p_args)
axes[1].plot(range(T+1), np.zeros(T+1), 'k-')

for ax in axes:
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=2, **legend_args)
plt.show()
```
Exercise 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.

```python
# == Model parameters == #
\( r = 0.05 \)
\( \beta = 1/(1 + r) \)
\( T = 60 \)
\( K = 40 \)
\( c_bar = 4 \)
\( \sigma = 0.35 \)
\( \mu = 4 \)
\( q = 1e4 \)
\( s = 1 \)
\( m_l = 2 \times \mu/K \)
```
m2 = -\mu/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=\beta, 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 = [[0],
     [0],
     [0],
     [0]]

# == Set up working life LQ instance with terminal Rf from lq_retired == #
lq_working = LQ(Q, R, A, B, C, beta=\beta, 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, :]

up = np.column_stack((up_w, up_r))
c = up.flatten() + c_bar

time = np.arange(1, K+1)
income_w = \sigma \ast wp_w[0, 1:K+1] + m1 * time + m2 * time**2
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))

bbox = (0., 1.02, 1., .102)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.7}
axes[0].plot(range(1, T+1), income, 'g-', label="non-financial income", **p_args)
axes[0].plot(range(T), c, 'k-', label="consumption", **p_args)
axes[1].plot(range(T+1), assets, 'b-', label="assets", **p_args)
axes[1].plot(range(T+1), np.zeros(T+1), 'k-')

for ax in axes:
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=2, **legend_args)

plt.show()
Exercise 3

The first task is to find the matrices $A, B, C, Q, R$ that define the LQ problem

Recall that $x_t = (\tilde{q}_t \ q_t \ 1)'$, 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 $\tilde{q}_t = m_0 + m_1 d_t$, and then, with some manipulation

$$\tilde{q}_{t+1} = m_0 (1 - \rho) + \rho \tilde{q}_t + m_1 \sigma u_{t+1}$$

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 conve-
nience:

\[
\min \mathbb{E}\left\{ \sum_{t=0}^{\infty} \beta^t a_1(q_t - \bar{q}_t)^2 + \gamma u_t^2 \right\}
\]

Our solution code is

```python
# == Model parameters == #
a0 = 5
a1 = 0.5
sigma = 0.15
rho = 0.9
gamma = 1
beta = 0.95
c = 2
T = 120

# == Useful constants == #
m0 = (a0-c)/(2 * a1)
m1 = 1/(2 * a1)

# == Formulate LQ problem == #
Q = gamma
R = [[a1, -a1, 0],
     [-a1, a1, 0],
     [0, 0, 0]]
A = [[rho, 0, m0 * (1 - rho)],
     [0, 1, 0],
     [0, 0, 1]]
B = [[0],
     [1],
     [0]]
C = [[m1 * sigma],
     [0],
     [0]]
lq = LQ(Q, R, A, B, C=C, beta=beta)

# == 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., 1.01)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.6}
time = range(len(q))
```
6.12 Optimal Savings I: The Permanent Income Model

Contents

- Optimal Savings I: The Permanent Income Model
  - Overview
  - The Savings Problem
  - Alternative Representations
6.12.1 Overview

This lecture describes a rational expectations version of the famous permanent income model of Milton Friedman [Fri56].

Robert Hall cast Friedmans model within a linear-quadratic setting [Hal78].

Like Hall, we formulate an infinite-horizon linear-quadratic savings problem.

We use the model as a vehicle for illustrating

- alternative formulations of the state of a dynamic system
- the idea of cointegration
- impulse response functions
- the idea that changes in consumption are useful as predictors of movements in income

Background readings on the linear-quadratic-Gaussian permanent income model are Halls [Hal78] and chapter 2 of [LS18].

6.12.2 The Savings Problem

In this section we state and solve the savings and consumption problem faced by the consumer.

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

\[
E_t[X_{t+1}] = X_t, \quad t = 0, 1, 2, \ldots
\]

Here \( E_t := E[ \cdot | \mathcal{F}_t] \) 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).

**The Decision Problem**

A consumer has preferences over consumption streams that are ordered by the utility functional

\[
E_0 \left[ \sum_{t=0}^{\infty} \beta^t u(c_t) \right] \quad (6.89)
\]

where

- \( E_t \) 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 (6.89) by choosing a consumption, borrowing plan \( \{c_t, b_{t+1}\}_{t=0}^{\infty} \) subject to the sequence of budget constraints

\[
c_t + b_t = \frac{1}{1 + r} b_{t+1} + y_t \quad t \geq 0 \quad (6.90)
\]

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.
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

\[
    z_{t+1} = Az_t + Cw_{t+1} \\
    y_t = Uz_t
\]  

(6.91)

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 \(\gamma\) 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

\[
    \mathbb{E}_0 \left[ \sum_{t=0}^{\infty} \beta^t b_t^2 \right] < \infty 
\]  

(6.92)

This condition rules out an always-borrow scheme that would allow the consumer to enjoy bliss consumption forever.

First-Order Conditions

First-order conditions for maximizing (6.89) subject to (6.90) are

\[
    \mathbb{E}_t [u'(c_t+1)] = u'(c_t), \quad t = 0, 1, \ldots
\]  

(6.93)

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, (6.93) has the striking implication that consumption follows a martingale:

\[ \mathbb{E}_t[c_{t+1}] = c_t \]  

(6.94)

(In fact quadratic preferences are necessary for this conclusion\(^1\))

One way to interpret (6.94) is that consumption will change only when new information about permanent income is revealed.

These ideas will be clarified below.

**The Optimal Decision Rule**

Now let’s deduce the optimal decision rule\(^2\)

**Note:** One way to solve the consumers 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 (6.94)
2. the period-by-period budget constraint (6.90), and
3. the boundary condition (6.92)

To accomplish this, observe first that (6.92) implies \( \lim_{t \to \infty} \beta^\frac{1}{2} b_{t+1} = 0 \)

Using this restriction on the debt path and solving (6.90) forward yields

\[ b_t = \sum_{j=0}^{\infty} \beta^j (y_{t+j} - c_{t+j}) \]  

(6.95)

Take conditional expectations on both sides of (6.95) and use the martingale property of consumption and the *law of iterated expectations* to deduce

\[ b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - \frac{c_t}{1 - \beta} \]  

(6.96)

---

\(^1\) A linear marginal utility is essential for deriving (6.94) from (6.93). 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 (6.93). 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 (6.93) 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 consumers savings also diverge to \( +\infty \).

\(^2\) An optimal decision rule is a map from current state into current actions in this case, consumption...
Expressed in terms of $c_t$ we get

$$c_t = (1 - \beta) \left[ \sum_{j=0}^{\infty} \beta^j E_t[y_{t+j}] - b_t \right] = \frac{r}{1 + r} \left[ \sum_{j=0}^{\infty} \beta^j E_t[y_{t+j}] - b_t \right]$$

(6.97)

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 $\sum_{j=0}^{\infty} \beta^j E_t[y_{t+j}]$
- **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 $[b_t \quad z_t]$

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 consumers 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

$$\sum_{j=0}^{\infty} \beta^j E_t[y_{t+j}] = E_t \left[ \sum_{j=0}^{\infty} \beta^j y_{t+j} \right] = U(I - \beta A)^{-1} z_t$$

Combining this with (6.97) gives

$$c_t = \frac{r}{1 + r} \left[ U(I - \beta A)^{-1} z_t - b_t \right]$$

(6.98)
Using this equality to eliminate $c_t$ in the budget constraint (6.90) gives

$$b_{t+1} = (1 + r)(b_t + c_t - y_t)$$

$$= (1 + r)b_t + r[U(I - \beta A)^{-1}z_t - b_t] - (1 + r)Uz_t$$

$$= b_t + U[r(I - \beta A)^{-1} - (1 + r)I]z_t$$

$$= b_t + U(I - \beta A)^{-1}(A - I)z_t$$

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:

$$z_{t+1} = Az_t + Cw_{t+1}$$

$$b_{t+1} = b_t + U[(I - \beta A)^{-1}(A - I)]z_t$$

$$y_t = Uz_t$$

$$c_t = (1 - \beta)[U(I - \beta A)^{-1}z_t - b_t]$$

(6.99)

To write this more succinctly, let

$$x_t = \begin{bmatrix} z_t \\ b_t \end{bmatrix}, \quad \tilde{A} = \begin{bmatrix} A \\ U(I - \beta A)^{-1}(A - I) \end{bmatrix}, \quad \tilde{C} = \begin{bmatrix} C' \\ 0 \end{bmatrix}$$

and

$$\tilde{U} = \begin{bmatrix} U \\ (1 - \beta)U(I - \beta A)^{-1} - (1 - \beta) \end{bmatrix}, \quad \tilde{y}_t = \begin{bmatrix} y_t \\ c_t \end{bmatrix}$$

Then we can express equation (6.99) as

$$x_{t+1} = \tilde{A}x_t + \tilde{C}w_{t+1}$$

$$\tilde{y}_t = \tilde{U}x_t$$

(6.100)

We can use the following formulas from linear state space models to compute population mean $\mu_t = \mathbb{E}x_t$ and covariance $\Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)^T]$

$$\mu_{t+1} = \tilde{A}\mu_t \quad \text{with} \quad \mu_0 \text{ given}$$

(6.101)

$$\Sigma_{t+1} = \tilde{A}\Sigma_t\tilde{A}^T + \tilde{C}\tilde{C}' \quad \text{with} \quad \Sigma_0 \text{ given}$$

(6.102)
We can then compute the mean and covariance of $\tilde{y}_t$ from

\begin{align*}
\mu_{y,t} &= \tilde{U} \mu_t \\
\Sigma_{y,t} &= \tilde{U} \Sigma_t \tilde{U}'
\end{align*}

(6.103)

**A Simple Example with iid Income**

To gain some preliminary intuition on the implications of (6.99), 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

\[
\begin{bmatrix}
    z_t \\
    1
\end{bmatrix}
, \quad
A = \begin{bmatrix}
    0 & 0 \\
    0 & 1
\end{bmatrix}
, \quad
U = \begin{bmatrix}
    1 & \mu
\end{bmatrix}
, \quad
C = \begin{bmatrix}
    \sigma \\
    0
\end{bmatrix}
\]

Finally, let \( b_0 = z_0 = 0 \)

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

\[
\begin{align*}
b_t &= -\sigma \sum_{j=1}^{t-1} w_j \\
c_t &= \mu + (1 - \beta)\sigma \sum_{j=1}^{t} w_j
\end{align*}
\]

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 \( \sigma = 0.15 \)

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

r = 0.05
beta = 1 / (1 + r)
T = 60
sigma = 0.15
mu = 1

def time_path():
    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):
```

\begin{verbatim}
    b[t] = w[1:t].sum()
    b = -\sigma * b
    c = \mu + (1 - \beta) * (\sigma * w - b)
    return w, b, c
\end{verbatim}

fig, ax = plt.subplots(figsize=(10, 6))

p_args = {'lw': 2, 'alpha': 0.7}
ax.grid()
ax.set_xlabel('Time')
bbox = (.0, 1.02, 1., .102)
legend_args = {'bbox_to_anchor': bbox, 'loc': 'upper left', 'mode': 'expand'}

w, b, c = time_path()
ax.plot(list(range(T+1)), \mu + \sigma * w, 'g-', label="non-financial income", **p_args)
ax.plot(list(range(T+1)), c, 'k-', label="consumption", **p_args)
ax.plot(list(range(T+1)), b, 'b-', label="debt", **p_args)
ax.legend(ncol=3, **legend_args)
plt.show()

Observe that consumption is considerably smoother than income.
The figure below shows the consumption paths of 250 consumers with independent income streams.
fig, ax = plt.subplots(figsize=(10, 6))
p_args = {'lw': 0.8, 'alpha': 0.7}
ax.grid()
ax.set_xlabel('Time')
ax.set_ylabel('Consumption')
b_sum = np.zeros(T+1)
for i in range(250):
    rcolor = random.choice(('c', 'g', 'b', 'k'))
    w, b, c = time_path()
    ax.plot(list(range(T+1)), c, color=rcolor, **p_args)
plt.show()

6.12.3 Alternative Representations

In this section we shed more light on the evolution of savings, debt and consumption by representing their dynamics in several different ways

Halls Representation

Hall [Hal78] suggested an insightful way to summarize the implications of LQ permanent income theory

First, to represent the solution for $b_t$, shift (6.97) forward one period and eliminate $b_{t+1}$ by using (6.90) 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

\[ c_{t+1} - c_t = (1 - \beta) \sum_{j=0}^{\infty} \beta^j \left\{ \mathbb{E}_{t+1}[y_{t+j+1}] - \mathbb{E}_t[y_{t+j+1}] \right\} \quad (6.104) \]

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 (6.104) and (6.96), which we repeat:

\[ b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - \frac{1}{1 - \beta} c_t \quad (6.105) \]

Equation (6.105) asserts that the consumers debt due at \( t \) equals the expected present value of its endowment minus the expected present value of its consumption stream

A high debt thus indicates a large expected present value of surpluses \( y_t - c_t \)

Recalling again our discussion on forecasting geometric sums, we have

\[ \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} = U (I - \beta A)^{-1} z_t \]
\[ \mathbb{E}_{t+1} \sum_{j=0}^{\infty} \beta^j y_{t+j+1} = U (I - \beta A)^{-1} z_{t+1} \]
\[ \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j+1} = U (I - \beta A)^{-1} A z_t \]

Using these formulas together with (6.91) and substituting into (6.104) and (6.105) gives the following representation for the consumers optimum decision rule:

\[ c_{t+1} = c_t + (1 - \beta) U (I - \beta A)^{-1} C w_{t+1} \]
\[ b_t = U (I - \beta A)^{-1} z_t - \frac{1}{1 - \beta} c_t \quad (6.106) \]
\[ y_t = U z_t \]
\[ z_{t+1} = A z_t + C w_{t+1} \]

Representation (6.106) makes clear that

- The state can be taken as \((c_t, z_t)\)

---

Chapter 6. Dynamic Programming
– 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 $(1 - \beta)U(I - \beta A)^{-1}Cw_{t+1}$
– This is a more explicit representation of the martingale result in (6.94)

**Cointegration**

Representation (6.106) reveals that the joint process \{c_t, b_t\} possesses the property that Engle and Granger [EG87] called cointegration

Cointegration is a tool that allows us to apply powerful results from the theory of stationary stochastic processes to (certain transformations of) nonstationary models

To apply cointegration in the present context, suppose that $z_t$ is asymptotically stationary\(^4\)

Despite this, both $c_t$ and $b_t$ will be non-stationary because they have unit roots (see (6.99) 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 (6.106) we have

$$(1 - \beta)b_t + c_t = (1 - \beta)U(I - \beta A)^{-1}z_t \quad (6.107)$$

Hence the linear combination $(1 - \beta)b_t + c_t$ is asymptotically stationary

Accordingly, Granger and Engle would call $[(1 - \beta) \quad 1]$ a **cointegrating vector** for the state

When applied to the nonstationary vector process $[b_t \quad c_t]'$, it yields a process that is asymptotically stationary

Equation (6.107) can be rearranged to take the form

$$(1 - \beta)b_t + c_t = (1 - \beta)E_t\sum_{j=0}^{\infty} \beta^j y_{t+j} \quad (6.108)$$

Equation (6.108) asserts that the cointegrating residual on the left side equals the conditional expectation of the geometric sum of future incomes on the right\(^6\)

**Cross-Sectional Implications**

Consider again (6.106), this time in light of our discussion of distribution dynamics in the lecture on linear systems

The dynamics of $c_t$ are given by

---

\(^4\) This would be the case if, for example, the spectral radius of $A$ is strictly less than one

\(^6\) See [JYC88], [LL01], [LLD04] for interesting applications of related ideas.
\[ c_{t+1} = c_t + (1 - \beta)U(I - \beta A)^{-1}Cw_{t+1} \]

or

\[ c_t = c_0 + \sum_{j=1}^{t} \hat{w}_j \quad \text{for} \quad \hat{w}_{t+1} := (1 - \beta)U(I - \beta A)^{-1}Cw_{t+1} \]

The unit root affecting \( c_t \) causes the time \( t \) variance of \( c_t \) to grow linearly with \( t \)

In particular, since \( \{\hat{w}_t\} \) is iid, we have

\[ \text{Var}[c_t] = \text{Var}[c_0] + t \sigma^2 \]

where

\[ \sigma^2 := (1 - \beta)^2U(I - \beta A)^{-1}CC'(I - \beta A')^{-1}U' \]

When \( \sigma > 0 \), \( \{c_t\} \) 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 (6.110) tells us that the variance of \( c_t \) increases over time at a rate proportional to \( t \)

A number of different studies have investigated this prediction and found some support for it (see, e.g., [DP94], [STY04])

**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 \( j \geq 1 \)

**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 (6.91) has the moving average representation

\[ y_{t+1} = d(L)w_{t+1} \]

where
• \(d(L) = \sum_{j=0}^{\infty} d_j L^j\) for some sequence \(d_j\), where \(L\) is the lag operator\(^3\)

• at time \(t\), the consumer has an information set\(^5\) \(w^t = [w_t, w_{t-1}, \ldots]\)

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

\[\mathbb{E}_{t+1}[y_{t+j}] - \mathbb{E}_t[y_{t+j}] = d_{j-1} w_{t+1}\] (6.112)

Using (6.112) in (6.104) gives

\[c_{t+1} - c_t = (1 - \beta)d(\beta)w_{t+1}\] (6.113)

The object \(d(\beta)\) is the present value of the moving average coefficients in the representation for the endowment process \(y_t\)

### 6.12.4 Two Classic Examples

We illustrate some of the preceding ideas with two examples

In both examples, the endowment follows the process \(y_t = z_{1t} + z_{2t}\) where

\[
\begin{bmatrix}
  z_{1t+1} \\
  z_{2t+1}
\end{bmatrix} = \begin{bmatrix}
  1 & 0 \\
  0 & 0
\end{bmatrix}\begin{bmatrix}
  z_{1t} \\
  z_{2t}
\end{bmatrix} + \begin{bmatrix}
  \sigma_1 & 0 \\
  0 & \sigma_2
\end{bmatrix}\begin{bmatrix}
  w_{1t+1} \\
  w_{2t+1}
\end{bmatrix}
\]

Here

• \(w_{t+1}\) is an iid \(2 \times 1\) process distributed as \(N(0, I)\)

• \(z_{1t}\) is a permanent component of \(y_t\)

• \(z_{2t}\) is a purely transitory component of \(y_t\)

#### Example 1

Assume as before that the consumer observes the state \(z_t\) at time \(t\)

In view of (6.106) we have

\[c_{t+1} - c_t = \sigma_1 w_{1t+1} + (1 - \beta)\sigma_2 w_{2t+1}\] (6.114)

Formula (6.114) shows how an increment \(\sigma_1 w_{1t+1}\) to the permanent component of income \(z_{1t+1}\) leads to

---

\(^3\) Representation (6.91) implies that \(d(L) = U(1 - AL)^{-1}C\).

\(^5\) 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.

---

• 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 \(\beta\) is saved, leading to a permanent increment in \(-b_{t+1}\)

Application of the formula for debt in (6.99) to this example shows that

\[
b_{t+1} - b_t = -z_{2t} = -\sigma_2 w_{2t}
\]

(6.115)

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.

```python
r = 0.05
\beta = 1 / (1 + r)
T = 20  # Time horizon
S = 5   # Impulse date
\sigma_1 = \sigma_2 = 0.15

def time_path(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] - \sigma_2 * w2[t]
        c[t+1] = c[t] + \sigma_1 * w1[t+1] + (1 - \beta) * \sigma_2 * w2[t+1]
    return b, c

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
p_args = {'lw': 2, 'alpha': 0.7}
titles = ['transitory', 'permanent']
L = 0.175

for ax, truefalse, title in zip(axes, (True, False), titles):
    b, c = time_path(permanent=truefalse)
    ax.set_title(f'Impulse response: {title} income shock')
    ax.plot(list(range(T+1)), c, 'g-', label="consumption", **p_args)
    ax.plot(list(range(T+1)), b, 'b-', label="debt", **p_args)
    ax.plot((S, S), (-L, L), 'k-', lw=0.5)
```

Chapter 6. Dynamic Programming
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 (6.106)

The discussion in sections 2.9.1 and 2.11.3 of [LS18] shows that the pertinent state space representation for
\[ y_t \text{ is} \]
\[
\begin{bmatrix}
    y_{t+1} \\
    a_{t+1}
\end{bmatrix} = 
\begin{bmatrix}
    1 & -(1-K) \\
    0 & 0
\end{bmatrix} 
\begin{bmatrix}
    y_t \\
    a_t
\end{bmatrix} + 
\begin{bmatrix}
    1 \\
    1
\end{bmatrix} a_{t+1}
\]
\[
y_t = 
\begin{bmatrix}
    1 & 0
\end{bmatrix} 
\begin{bmatrix}
    y_t \\
    a_t
\end{bmatrix}
\]

where

- \( K := \text{the stationary Kalman gain} \)
- \( a_t := y_t - E[y_t | y_{t-1}, \ldots, y_0] \)

In the same discussion in [LS18] it is shown that \( K \in [0, 1] \) and that \( K \) increases as \( \sigma_1/\sigma_2 \) 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 (6.106) implies

\[
c_{t+1} - c_t = [1 - \beta(1 - K)]a_{t+1} \tag{6.116}
\]

where the endowment process can now be represented in terms of the univariate innovation to \( y_t \) as

\[
y_{t+1} - y_t = a_{t+1} - (1 - K)a_t \tag{6.117}
\]

Equation (6.117) 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 (6.117), the first difference of income is a first-order moving average.

Equation (6.116) asserts that the first difference of consumption is iid.

Application of formula to this example shows that

\[
b_{t+1} - b_t = (K - 1)a_t \tag{6.118}
\]

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.
6.12.5 Further Reading

The model described above significantly changed how economists think about consumption

While Hall’s model does a remarkably good job as a first approximation to consumption data, it is 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., [HMS82], [Par99], [Dea91], [Car01]

6.12.6 Appendix: the Euler Equation

Where does the first order condition (6.93) come from?

Here we 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 (6.90) 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 (6.93) in the two period case

The proof for the general case is similar

6.13 Optimal Savings II: LQ Techniques

Contents

- Optimal Savings II: LQ Techniques
  - Overview
  - Setup
  - The LQ Approach
  - Implementation
  - Two Example Economies
6.13.1 Overview

This lecture continues our analysis of the linear-quadratic (LQ) permanent income model of savings and consumption.

As we saw in our previous lecture on this topic, Robert Hall [Hal78] used the LQ permanent income model to restrict and interpret intertemporal comovements of nondurable consumption, nonfinancial income, and financial wealth.

For example, we saw how the model asserts that for any covariance stationary process for nonfinancial income,

- consumption is a random walk
- financial wealth has a unit root and is cointegrated with consumption

Other applications use the same LQ framework.

For example, a model isomorphic to the LQ permanent income model has been used by Robert Barro [Bar79] to interpret intertemporal comovements of a government's tax collections, its expenditures net of debt service, and its public debt.

This isomorphism means that in analyzing the LQ permanent income model, we are in effect also analyzing the Barro tax smoothing model.

It is just a matter of appropriately relabeling the variables in Hall's model.

In this lecture, we

- 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

Well then use these characterizations to construct a simple model of cross-section wealth and consumption dynamics in the spirit of Truman Bewley [Bew86].

(Later we will 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

6.13.2 Setup

Let's recall the basic features of the model discussed in permanent income model.

Consumer preferences are ordered by
\[
E_0 \sum_{t=0}^{\infty} \beta^t u(c_t) \quad (6.119)
\]

where \( u(c) = -(c - \gamma)^2 \)

The consumer maximizes (6.119) by choosing a consumption, borrowing plan \( \{c_t, b_{t+1}\}_{t=0}^{\infty} \) subject to the sequence of budget constraints

\[
c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t, \quad t \geq 0 \quad (6.120)
\]

and the no-Ponzi condition

\[
E_0 \sum_{t=0}^{\infty} \beta^t b_t^2 < \infty \quad (6.121)
\]

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

\[
\begin{align*}
z_{t+1} &= Az_t + Cw_{t+1} \\
y_t &= Uz_t \quad (6.122)
\end{align*}
\]

The restrictions on the shock process and parameters are the same as in our previous lecture

**Digression on a useful isomorphism**

The LQ permanent income model of consumption is mathematically isomorphic with a version of Barros [Bar79] model of tax smoothing.

In the LQ permanent income model

- the household faces an exogenous process of nonfinancial income
- the household wants to smooth consumption across states and time

In the Barro tax smoothing model

- a government faces an exogenous sequence of government purchases (net of interest payments on its debt)
- a government wants to smooth tax collections across states and time

If we set

- \( T_t \), total tax collections in Barros model to consumption \( c_t \) in the LQ permanent income model
• $G_t$, exogenous government expenditures in Barros model to nonfinancial income $y_t$ in the permanent income model

• $B_t$, government risk-free one-period assets falling due in Barros model to risk-free one period consumer debt $b_t$ falling due in the LQ permanent income model

• $R_t$ the gross rate of return on risk-free one-period government debt in Barros 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

**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 (6.122), as discussed in our lecture on linear models.

To do so we take

$$z_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ \sigma \\ 0 \end{bmatrix}, \quad \text{and} \quad U = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

**6.13.3 The LQ Approach**

*Previously* we solved the permanent income model by solving a system of linear expectational difference equations subject to two boundary conditions

Here we solve the same model using *LQ methods* based on dynamic programming

After confirming that answers produced by the two methods agree, we apply QuantEcon’s LinearStateSpace class to illustrate features of the model

Why solve a model in two distinct ways?

Because by doing so we gather insights about the structure of the model

Our earlier approach based on solving a system of expectational difference equations brought to the fore the role of the consumers 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

682 Chapter 6. Dynamic Programming
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

$$
E \sum_{t=0}^{\infty} \beta^t \{x_t'Rx_t + u_t'Qu_t\},
$$

subject to $x_0$ given and the law of motion

$$
x_{t+1} = \tilde{A}x_t + \tilde{B}u_t + \tilde{C}w_{t+1}, \quad t \geq 0,
$$

where $w_{t+1}$ is iid with mean vector zero and $Ew_tw_t' = I$

The tildes in $\tilde{A}, \tilde{B}, \tilde{C}$ are to avoid clashing with notation in (6.122)

The value function for this problem is $v(x) = -x'Px - d$, 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 $u_t = -Fx_t$, where $F := \beta(Q + \beta\tilde{B}'\tilde{P}\tilde{B})^{-1}\tilde{B}'P\tilde{A}$

Under an optimal decision rule $F$, the state vector $x_t$ evolves according to $x_{t+1} = (\tilde{A} - \tilde{B}F)x_t + \tilde{C}w_{t+1}$

Mapping into the LQ framework

To map into the LQ framework, well 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 (6.123) 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 (6.123) match the dynamics of $z_t$ and $b_t$ described above

To map utility into the quadratic form $x_t'Rx_t + u_t'Qu_t$ we can set

- $Q := 1$ (remember that we are minimizing) and
- $R := a 4 \times 4$ matrix of zeros
However, there is one problem remaining
We have no direct way to capture the non-recursive restriction (6.121) on the debt sequence \( \{b_t\} \) from within the LQ framework
To try to enforce it, were 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
Well check whether it really is small numerically soon

### 6.13.4 Implementation

Lets write some code to solve the model
One comment before we start is that the bliss level of consumption \( \gamma \) 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

#### The exogenous noinfinancial income process

First we create the objects for the optimal linear regulator

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

# Set parameters
alpha, beta, rho1, rho2, sigma = 10.0, 0.95, 0.9, 0.0, 1.0

R = 1 / beta
A = np.array([[1., 0., 0.],
              [alpha, rho1, rho2],
              [0., 1., 0.]]).T
C = np.array([[0.0], [sigma], [0.0]])
G = np.array([[0.0], [1.0], [0.0]])

# Form LinearStateSpace system and pull off steady state moments
mu_z0 = np.array([[1.0], [0.0], [0.0]])
Sigma_z0 = np.zeros((3, 3))
Lz = qe.LinearStateSpace(A, C, G, mu_0=mu_z0, Sigma_0=Sigma_z0)
mu_z, mu_y, Sigma_z, Sigma_y = Lz.stationary_distributions()

# Mean vector of state for the savings problem
mzo = np.vstack(([mu_z, 0.0]))
```

684 Chapter 6. Dynamic Programming
# 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.
mbewley = mxo
sxbewley = sxo

The next step is to create the matrices for the LQ system

A12 = np.zeros((3, 1))
ALQ_l = np.hstack([A, A12])
ALQ_r = np.array([[0, -R, 0, R]])
ALQ = np.vstack([ALQ_l, ALQ_r])

RLQ = np.array([[0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 1e-9]])

QLQ = np.array([1.0])
BLQ = np.array([0., 0., 0., R]).reshape(4, 1)
CLQ = np.array([0., σ, 0., 0.]).reshape(4, 1)

β_LQ = β

Lets print these out and have a look at them

print(f"A = 
{ALQ}"
print(f"B = 
{BLQ}"
print(f"R = 
{RLQ}"
print(f"Q = 
{QLQ}"

A =

[[ 1.  0.  0.  0. ]
 [ 10.  0.9 0.  0. ]
 [ 0.  1.  0.  0. ]
 [ 0. -1.0526 0.  1.0526]]

B =

[[ 0. ]
 [ 0. ]
 [ 0. ]
 [ 1.0526]]

R =

[[ 0.  0.  0.  0. ]
 [ 0.  0.  0.  0. ]
 [ 0.  0.  0.  0. ]
 [ 0.  0.  0.  0.]]

Q =

6.13. Optimal Savings II: LQ Techniques
Now create the appropriate instance of an LQ model

\[
\text{LQPI} = \text{qe.LQ}(\text{QLQ}, \text{RLQ}, \text{ALQ}, \text{BLQ}, \text{C=CLQ}, \text{beta=\beta}_{LQ})
\]

Well save the implied optimal policy function soon compare them with what we get by employing an alternative solution method

\[
\text{P, F, d} = \text{LQPI.stationary_values()} \quad \# \text{Compute value function and decision rule}
\]

\[
\text{ABF} = \text{ALQ} - \text{BLQ} @ \text{F} \quad \# \text{Form closed loop system}
\]

**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

\[
\begin{align*}
z_{t+1} &= Az_t + Cw_{t+1} \\
b_{t+1} &= b_t + U[(I - \beta A)^{-1}(A - I)]z_t \\
y_t &= Uz_t \\
c_t &= (1 - \beta)[U(I - \beta A)^{-1}z_t - b_t]
\end{align*}
\]

Now we apply the formulas in this system

\[
\begin{align*}
\text{b_pol} &= G@\text{la.inv}(\text{np.eye(3, 3)} - \beta * \text{A}) @ (\text{A} - \text{np.eye(3, 3)}) \\
c_{\text{pol}} &= (1 - \beta) * G@\text{la.inv}(\text{np.eye(3, 3)} - \beta * \text{A})
\end{align*}
\]

\[
\begin{align*}
\text{A}_\text{LSS1} &= \text{np.vstack([A, b_pol])} \\
\text{A}_\text{LSS2} &= \text{np.eye(4, 1, -3)} \\
\text{A}_\text{LSS} &= \text{np.hstack([A}_\text{LSS1}, \text{A}_\text{LSS2})}
\end{align*}
\]

\[
\begin{align*}
\text{C}_\text{LSS} &= \text{np.vstack([C, np.zeros(1)])}
\end{align*}
\]

\[
\begin{align*}
\text{G}_\text{LSS1} &= \text{np.vstack([G, c_{\text{pol}}])} \\
\text{G}_\text{LSS2} &= \text{np.vstack([np.zeros(1), -(1 - \beta)])} \\
\text{G}_\text{LSS} &= \text{np.hstack([G}_\text{LSS1}, \text{G}_\text{LSS2])}
\end{align*}
\]
# Use the following values to start everyone off at b=0, initial incomes zero

\[
\mu_0 = \text{np.array([1., 0., 0., 0.])}
\]

\[
\Sigma_0 = \text{np.zeros((4, 4))}
\]

A\_LSS calculated as we have here should equal ABF calculated above using the LQ model

\[
ABF - A\_LSS
\]

\[
\text{array([[ 0. , 0. , 0. , 0. ],
          [ 0. , 0. , 0. , 0. ],
          [ 0. , 0. , 0. , 0. ],
          [-0.0001, -0. , 0. , 0. ]])}
\]

Now compare pertinent elements of \( c\_pol \) and \( F \)

\[
\text{print}(c\_pol, "\n", -F)
\]

\[
[[ 65.5172 0.3448 0. ]]
[[ 65.5172 0.3448 -0. -0.05 ]]
\]

We have verified that the two methods give the same solution

Now lets create instances of the LinearStateSpace class and use it to do some interesting experiments

To do this, well use the outcomes from our second method

### 6.13.5 Two Example Economies

In the spirit of Bewley models [Bew86], well 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 \( \text{ex ante} \) 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 \( \text{ex ante} \) 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
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 a first set of graphs, darker lines depict a particular sample path, while the lighter lines describe 24 other paths.

A second graph plots a collection of simulations against the population distribution that we extract from the `LinearStateSpace` instance LSS.

Comparing sample paths with population distributions at each date $t$ is a useful exercise – see our discussion of the laws of large numbers.

```
LSS = qe.LinearStateSpace(A_LSS, C_LSS, G_LSS, mu_0=μ_0, Sigma_0=Σ_0)
```

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

```python
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 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)
\[
\begin{align*}
    d_{90} &= 1.65 \times \text{np.sqrt(debt\_var)} \\
    d_{95} &= 1.96 \times \text{np.sqrt(debt\_var)} \\
    d_{\text{perc}\_95p}, d_{\text{perc}\_95m} &= \text{debt\_mean} + d_{95}, \text{debt\_mean} - d_{95} \\
    d_{\text{perc}\_90p}, d_{\text{perc}\_90m} &= \text{debt\_mean} + d_{90}, \text{debt\_mean} - d_{90}
\end{align*}
\]

# Create second figure
fig, ax = plt.subplots(2, 1, figsize=(10, 8))

# 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 lets create figures with initial conditions of zero for \(y_0\) and \(b_0\)

\[
\begin{align*}
    \text{out} &= \text{income\_consumption\_debt\_series}(A\_LSS, C\_LSS, G\_LSS, \mu_0, \Sigma_0) \\
    \text{bsim0, csim0, ysim0} &= \text{out}[3:] \\
    \text{cons\_mean0, cons\_var0, debt\_mean0, debt\_var0} &= \text{out}[3:] \\
    \text{consumption\_income\_debt\_figure(bsim0, csim0, ysim0)}
\end{align*}
\]

plt.show()
consumption_debt_fanchart(csím0, cons_mean0, cons_var0,
    bsím0, 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

$$(1 - \beta)b_t + c_t = (1 - \beta)E_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \tag{6.124}$$

So at time 0 we have

$$c_0 = (1 - \beta)E_0 \sum_{t=0}^{\infty} \beta^t y_t$$

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 \textit{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 $y_{-1} = y_{-2} = 0$

\textbf{Cointegration residual}

The following figure plots realizations of the left side of (6.124), which, \textit{as discussed in our last lecture}, is called the \textbf{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 households 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 \textit{asymptotically} covariance stationary, not \textit{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

```python
def cointegration_figure(bsim, csim):
    r"""
    Plots the cointegration
    """

    # Create figure
    fig, ax = plt.subplots(figsize=(10, 8))
    ax.plot((1 - \beta) + bsim[:, :], csim[:, :], color="k")
    ax.plot((1 - \beta) + bsim.T + csim.T, color="k", alpha=0.1)

    ax.set(title="Cointegration of Assets and Consumption", xlabel="t")

    return fig

cointegration_figure(bsim0, csim0)
plt.show()
```

6.13. Optimal Savings II: LQ Techniques

693
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 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 mans Bewley [Bew86] model in the following way:

- as before, we start everyone at \( b_0 = 0 \)

- But instead of starting everyone at \( y_{-1} = y_{-2} = 0 \), we draw \( \begin{bmatrix} y_{-1} \\ y_{-2} \end{bmatrix} \) from the invariant distribution of
the \( \{y_t\} \) process

This rigs a closed economy in which people are borrowing and lending with each other at a gross risk-free interest rate of \( R = \beta^{-1} \)

Across the group of people being analyzed, risk-free loans are in zero excess supply

We have arranged primitives so that \( R = \beta^{-1} \) clears the market for risk-free loans at zero aggregate excess supply

So the risk-free loans are being made from one person to another within our closed set of agents

There is no need for foreigners to lend to our group

Let's have a look at the corresponding figures

```python
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()
```

![Nonfinancial Income, Consumption, and Debt](image)

6.13. Optimal Savings II: LQ Techniques
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
6.14 Consumption and Tax Smoothing with Complete and Incomplete Markets

Contents

- Consumption and Tax Smoothing with Complete and Incomplete Markets
  - Overview
  - Background
  - Model 1 (Complete Markets)
6.14.1 Overview

This lecture describes two types of consumption-smoothing and tax-smoothing models

- one is in the complete markets tradition of Lucas and Stokey [LS83]
- the other is in the incomplete markets tradition of Hall [Hal78] and Barro [Bar79]

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

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

Hall [Hal78] and Barro [Bar79] both assumed 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 gross interest rate on one period risk-free debt that equals $\beta^{-1}$, where $\beta \in (0, 1)$ is also a consumers intertemporal discount factor

Barro [Bar79] made an analogous assumption about the risk-free interest rate in a tax-smoothing model that we regard as isomorphic to Halls consumption-smoothing model

We maintain Hall and Barros 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 that we use in a complete markets model in the style of Lucas and Stokey [LS83]

While we are equally interested in consumption-smoothing and tax-smoothing models, for the most part we focus explicitly on consumption-smoothing versions of these models

But for each version of the consumption-smoothing model there is a natural tax-smoothing counterpart obtained simply by

- relabeling consumption as tax collections and nonfinancial income as government expenditures
- relabeling the consumers debt as the governments assets

For elaborations on this theme, please see Optimal Savings II: LQ Techniques and later parts of this lecture

Well consider two closely related alternative assumptions about the consumers exogenous nonfinancial income process (or in the tax-smoothing interpretation, the governments exogenous expenditure process):

- that it obeys a finite $N$ state Markov chain (setting $N = 2$ most of the time)
- 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
Well spend most of this lecture studying the finite-state Markov specification, but will briefly treat the linear state space specification before concluding.

**Relationship to Other Lectures**

This lecture can be viewed as a followup to *Optimal Savings II: LQ Techniques* and a warm up for a model of tax smoothing described in *Optimal Taxation with State-Contingent Debt*.

Linear-quadratic versions of the Lucas-Stokey tax-smoothing model are described in *Optimal Taxation in an LQ Economy*.

The key difference between those lectures and this one is:

- Here the decision maker takes all prices as exogenous, meaning that his decisions do not affect them.
- In *Optimal Taxation in an LQ Economy* and *Optimal Taxation with State-Contingent Debt*, the decision maker – the government in the case of these lectures – recognizes that his decisions affect prices.

So these later lectures are partly about how the government should manipulate prices of government debt.

**6.14.2 Background**

Outcomes in consumption-smoothing (or tax-smoothing) models emerge from two sources:

- a decision maker – a consumer in the consumption-smoothing model or a government in the tax-smoothing model – who wants to maximize an intertemporal objective function that expresses its preference for paths of consumption (or tax collections) that are smooth in the sense of not varying across time and Markov states.
- a set of trading opportunities that allow the optimizer to transform a possibly erratic nonfinancial income (or government expenditure) process into a smoother consumption (or tax collections) process by purchasing or selling financial securities.

In the complete markets version of the model, each period the consumer can buy or sell one-period ahead state-contingent securities whose payoffs depend on next periods realization of the Markov state.

In the two-state Markov chain case, there are two such securities each period.

In an $N$ state Markov state version of the model, $N$ such securities are traded each period.

These state-contingent securities are commonly called Arrow securities, after Kenneth Arrow who first theorized about them.

In the incomplete markets version of the model, the consumer can buy and sell only one security each period, a risk-free bond with gross return $\beta^{-1}$.

**Finite State Markov Income Process**

In each version of the consumption-smoothing model, nonfinancial income is governed by a two-state Markov chain (its easy to generalize this to an $N$ state Markov chain).
In particular, the \textit{state of the world} is given by $s_t$ that follows a Markov chain with transition probability matrix

$$P_{tj} = \mathbb{P}\{s_{t+1} = s_j \mid s_t = s_i\}$$

Nonfinancial income $\{y_t\}$ obeys

$$y_t = \begin{cases} 
\bar{y}_1 & \text{if } s_t = s_1 \\
\bar{y}_2 & \text{if } s_t = s_2 
\end{cases}$$

A consumer wishes to maximize

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(c_t)\right] \quad \text{where} \quad u(c_t) = -(c_t - \gamma)^2 \quad \text{and} \quad 0 < \beta < 1 \quad (6.125)$$

\textbf{Remark About Isomorphism}

We can regard these as Barro [Bar79] tax-smoothing models if we set $c_t = T_t$ and $G_t = y_t$, where $T_t$ is total tax collections and $\{G_t\}$ is an exogenous government expenditures process.

\textbf{Market Structure}

The two models differ in how effectively the market structure allows the 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

\textbf{6.14.3 Model 1 (Complete Markets)}

At each date $t \geq 0$, the consumer trades \textbf{one-period ahead Arrow securities}.

We assume that prices of these securities are exogenous to the consumer (or in the tax-smoothing version of the model, to the government).

\textit{Exogenous} means that they are unaffected by the decision maker.

In Markov state $s_t$ at time $t$, one unit of consumption in state $s_{t+1}$ at time $t+1$ costs $g(s_{t+1} \mid s_t)$ units of the time $t$ consumption good.

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 consumers budget constraint at $t \geq 0$ in Markov state $s_t$ is

$$c_t + b_t \leq y(s_t) + \sum_j q(s_j \mid s_t) b_{t+1}(s_j \mid s_t)$$

where $b_t$ is the consumers one-period debt that falls due at time $t$ and $b_{t+1}(s_j \mid s_t)$ are the consumers time $t$ sales of the time $t + 1$ consumption good in Markov state $s_j$, a source of time $t$ revenues.

An analogue of Halls assumption that the one-period risk-free gross interest rate is $\beta^{-1}$ is

$$q(s_j \mid s_t) = \beta P_{ij}$$  \hspace{1cm} (6.126)

To understand this, observe that in state $s_i$ it costs $\sum_j q(s_j \mid s_i)$ 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

$$\sum_j q(s_j \mid s_i) = \sum_j \beta P_{ij} = \beta$$

This confirms that (6.126) is a natural analogue of Halls assumption about the risk-free one-period interest rate.

First-order necessary conditions for maximizing the consumers expected utility are

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

or, under our assumption (6.126) on Arrow security prices,

$$c_{t+1} = c_t$$  \hspace{1cm} (6.127)

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.

Guess: Well make the plausible guess that

$$b_{t+1}(s_j \mid s_t = s_i) = b(s_j), \hspace{0.5cm} i = 1, 2; \hspace{0.5cm} j = 1, 2$$  \hspace{1cm} (6.128)

so that the amount borrowed today turns out to depend only on tomorrow’s Markov state. (Why is this a plausible guess?)

To determine $\bar{c}$, we shall pursue the implications of the consumers budget constraints in each Markov state today and our guess (6.128) about the consumers debt level choices.

For $t \geq 1$, these imply

$$\bar{c} + b(s_1) = y(s_1) + q(s_1 \mid s_1)b(s_1) + q(s_2 \mid s_1)b(s_2)$$

$$\bar{c} + b(s_2) = y(s_2) + q(s_1 \mid s_2)b(s_1) + q(s_2 \mid s_2)b(s_2),$$  \hspace{1cm} (6.129)
or

\[
\begin{bmatrix}
  b(\bar{s}_1) \\
  b(\bar{s}_2)
\end{bmatrix} + \begin{bmatrix}
  \bar{c} \\
  \bar{c}
\end{bmatrix} = \begin{bmatrix}
  y(\bar{s}_1) \\
  y(\bar{s}_2)
\end{bmatrix} + \beta \begin{bmatrix}
  P_{11} & P_{12} \\
  P_{21} & P_{22}
\end{bmatrix} \begin{bmatrix}
  b(\bar{s}_1) \\
  b(\bar{s}_2)
\end{bmatrix}
\]

These are 2 equations in the 3 unknowns $\bar{c}, b(\bar{s}_1), b(\bar{s}_2)$

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 is $\bar{s}_1$

Then the budget constraint at time $t = 0$ is

\[
\bar{c} + b_0 = y(\bar{s}_1) + q(\bar{s}_1 | \bar{s}_1) b(\bar{s}_1) + q(\bar{s}_2 | \bar{s}_1) b(\bar{s}_2)
\]  \tag{6.130}

If we substitute (6.130) into the first equation of (6.129) and rearrange, we discover that

\[
b(\bar{s}_1) = b_0
\]  \tag{6.131}

We can then use the second equation of (6.129) to deduce the restriction

\[
y(\bar{s}_1) - y(\bar{s}_2) + [q(\bar{s}_1 | \bar{s}_1) - q(\bar{s}_1 | \bar{s}_2) - 1] b_0 + [q(\bar{s}_2 | \bar{s}_1) + 1 - q(\bar{s}_2 | \bar{s}_2)] b(\bar{s}_2) = 0,
\]  \tag{6.132}

an equation in the unknown $b(\bar{s}_2)$

Knowing $b(\bar{s}_1)$ and $b(\bar{s}_2)$, we can solve equation (6.130) for the constant level of consumption $\bar{c}$

**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 Markov states

We 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 consumers debt neither accumulates, nor decumulates, nor drifts. Instead the debt level each period is an exact function of the Markov state, so in the two-state Markov case, it switches between two values

- We have verified guess (6.128)

We computed how one of those debt levels depends entirely on initial debt – it equals it – and how the other value depends on virtually all remaining parameters of the model
Here’s some code that, among other things, contains a function called `consumption_complete()`.

This function computes $b(s_1), b(s_2), \bar{c}$ as outcomes given a set of parameters, under the assumption of complete markets.

```python
import numpy as np
import quantecon as qe
import scipy.linalg as la

class ConsumptionProblem:
    def __init__(self, 
                 \beta=.96, 
                 y=[2, 1.5], 
                 b0=3, 
                 P=np.asarray([[.8, .2], 
                                [.4, .6]]):
        
        Parameters
        -------
        \beta : discount factor
        P : 2x2 transition matrix
        y : list containing the two income levels
        b0 : debt in period 0 (= state_1 debt level)
        
        self.\beta = \beta
        self.y = y
        self.b0 = b0
        self.P = P

    def consumption_complete(cp):
        
        Computes endogenous values for the complete market case.
        
        Parameters
        -------
        cp : instance of ConsumptionProblem
        
        Returns
        -------
        c_bar : constant consumption
        b1 : rolled over b0
```
b2 : debt in state_2

associated with the price system

\[ Q = \beta \ast P \]

""
\[ \beta, P, y, b0 = cp.\beta, cp.P, cp.y, cp.b0 \]
# Unpack

y1, y2 = y
# extract income levels
b1 = b0
# b1 is known to be equal to b0
Q = \beta \ast P
# assumed price system

# Using equation (7) calculate b2
b2 = (y2 - y1 - (Q[0, 0] - Q[1, 0] - 1) \ast b1) / (Q[0, 1] + 1 - Q[1, 1])

# Using equation (5) calculate c_bar
\[ c_{\bar{b}} = y1 - b0 + Q[0, :] \] @ np.asarray([b1, b2])

return c_bar, b1, b2

def consumption_incomplete(cp, N_simul=150):
""
Computes endogenous values for the incomplete market case.

Parameters
----------

cp : instance of ConsumptionProblem
N_simul : int
""

\[ \beta, P, y, b0 = cp.\beta, cp.P, cp.y, cp.b0 \]
# Unpack

# For the simulation define a quantecon MC class
mc = qe.MarkovChain(P)

# Useful variables
y = np.asarray(y).reshape(2, 1)
v = np.linalg.inv(np.eye(2) - \beta \ast P) @ y

# Simulat state path
s_path = mc.simulate(N_simul, init=0)

# 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)
\[ db = ((1 - \beta) \ast v - y) / \beta \]

for i, s in enumerate(s_path):
c_path[i] = (1 - \beta) \times (v - b_path[i] \times \text{np.ones((2, 1))))[s, 0]
b_path[i + 1] = b_path[i] + \text{db}[s, 0]

return c_path, b_path[:-1], y[s_path], s_path

Let's test by checking that \( \bar{c} \) and \( b_2 \) satisfy the budget constraint

```python
cp = ConsumptionProblem()
c_bar, b1, b2 = consumption_complete(cp)
debt_complete = np.asarray([b1, b2])
np.isclose(c_bar + b2 - cp.y[1] - (cp.\beta \times cp.P)[1, :] @ debt_complete, 0)
```

True

Below, we'll take the outcomes produced by this code—in particular the implied consumption and debt paths—and compare them with outcomes from an incomplete markets model in the spirit of Hall [Hal78] and Barro [Bar79] (and also, for those who love history, Gallatin (1807) [Gal37])

### 6.14.4 Model 2 (One-Period Risk Free Debt Only)

This is a version of the original models of Hall (1978) and Barro (1979) in which the decision maker's ability to substitute intertemporally is constrained by his ability to buy or sell only one security, a risk-free one-period bond bearing a constant gross interest rate that equals \( \beta^{-1} \).

Given an initial debt \( b_0 \) at time 0, the consumer faces a sequence of budget constraints

\[
c_t + b_t = y_t + \beta b_{t+1}, \quad t \geq 0
\]

where \( \beta \) is the price at time \( t \) of a risk-free claim on one unit of time consumption at time \( t + 1 \).

First-order conditions for the consumer's problem are

\[
\sum_j u'(c_{t+1,j})P_{ij} = u'(c_{t,i})
\]

For our assumed quadratic utility function this implies

\[
\sum_j c_{t+1,j}P_{ij} = c_{t,i}, \quad (6.133)
\]

which 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

\[
b_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - (1 - \beta)^{-1} c_t \quad (6.134)
\]
and

\[ c_t = (1 - \beta) \left[ E_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - b_t \right]. \] (6.135)

Equation (6.135) expresses \( c_t \) as a net interest rate factor \( 1 - \beta \) times the sum of the expected present value of nonfinancial income \( E_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \) and financial wealth \(-b_t\).

Substituting (6.135) into the one-period budget constraint and rearranging leads to

\[ b_{t+1} - b_t = \beta^{-1} \left( (1 - \beta) E_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - y_t \right) \] (6.136)

Now let's do a useful calculation that will yield a convenient expression for the key term \( E_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \) in our finite Markov chain setting.

Define

\[ v_t := E_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \]

In our finite Markov chain setting, \( v_t = v(1) \) when \( s_t = s_1 \) and \( v_t = v(2) \) when \( s_t = s_2 \).

Therefore, we can write

\[ v(1) = y(1) + \beta P_{11} v(1) + \beta P_{12} v(2) \]
\[ v(2) = y(2) + \beta P_{21} v(1) + \beta P_{22} v(2) \]

or

\[ \vec{v} = \vec{y} + \beta P \vec{v} \]

where \( \vec{v} = \begin{bmatrix} v(1) \\ v(2) \end{bmatrix} \) and \( \vec{y} = \begin{bmatrix} y(1) \\ y(2) \end{bmatrix} \).

We can also write the last expression as

\[ \vec{v} = (I - \beta P)^{-1} \vec{y} \]

In our finite Markov chain setting, from expression (6.135), consumption at date \( t \) when debt is \( b_t \) and the Markov state today is \( s_t = i \) is evidently

\[ c(b_t, i) = (1 - \beta) \left( (I - \beta P)^{-1} \vec{y} \right)_i - b_t \] (6.137)

and the increment in debt is

\[ b_{t+1} - b_t = \beta^{-1} [(1 - \beta) v(i) - y(i)] \] (6.138)
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 consumers 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 consumers debt and the dependence of current consumption on todays debt level account for the drift over time in consumption

The Incomplete Markets Model

The code above also contains a function called `consumption_incomplete()` that uses (6.137) and (6.138) to

- simulate paths of $y_t, c_t, b_{t+1}$
- plot these against values of 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

```python
import matplotlib.pyplot as plt
np.random.seed(1)
N_simul = 150
cp = ConsumptionProblem()

c_bar, b1, b2 = consumption_complete(cp)
debt_complete = np.asarray([b1, b2])

c_path, debt_path, y_path, s_path = consumption_incomplete(cp, N_simul=N_simul)

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')
```
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 it takes a random walk in the incomplete markets version of the model
- the consumers' debt oscillates between two values that are functions of the Markov state in the complete markets model, while the consumers' debt drifts in a unit root fashion in the incomplete markets economy

**Using the Isomorphism**

We can simply relabel variables to acquire tax-smoothing interpretations of our two models

```python
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')
```
6.14.5 Example: Tax Smoothing with Complete Markets

It is useful to focus on a simple tax-smoothing example with complete markets. This example will illustrate how, in a complete markets model like that of Lucas and Stokey [LS83], the government purchases insurance from the private sector.

- Purchasing insurance protects the government against the need to raise taxes too high or issue too much debt in the high government expenditure event.

We assume that government expenditures move between 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 of one unit of output next period in state $j$ when today's Markov state is $i$ and $b_i$ is the government's level of assets in Markov state $i$.

That is, $b_i$ is the amount of the one-period loans owned by the government that fall due at time $t$.

As above, we'll assume that the initial Markov state is state 1.

In addition, to simplify our example, we'll set the government's initial asset level to 0, so that $b_1 = 0$.

Here's our code to compute a quantitative example with zero debt in peace time:
# Parameters

\[ \beta = .96 \]
\[ y = [1, 2] \]
\[ b0 = 0 \]
\[ P = \text{np.asarray}([[.8, .2],
                          [.4, .6]]) \]

\[ cp = \text{ConsumptionProblem}(\beta, y, b0, P) \]
\[ Q = \beta \times P \]
\[ N_{\text{simul}} = 150 \]

c_bar, b1, b2 = consumption_complete(cp)
debt_complete = np.asarray([b1, b2])

print(f"P
\n{P}")
print(f"Q
\n{Q}")
print(f"Govt expenditures in peace and war = \(y\)")
print(f"Constant tax collections = \{c_bar\}\)
print(f"Govt assets in two states = \{debt_complete\}\)

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, 1] \times b2
print(f"Spending on Arrow war security in peace = \{AS1\}\)
AS2 = Q[1, 1] \times b2
print(f"Spending on Arrow war security in war = \{AS2\}\)

print("\n")
print("Government tax collections plus asset levels in peace and war")
TB1 = c_bar + b1
print(f"T+b in peace = \{TB1\}\)
TB2 = c_bar + b2
print(f"T+b in war = \{TB2\}\)

print("\n")
print("Total government spending in peace and war")
G1 = y[0] + AS1
G2 = y[1] + AS2
print(f"Peace = \{G1\}\)
print(f"War = \{G2\}\)

print("\n")
print("Let's see ex post and ex ante returns on Arrow securities")

\[ \Pi = \text{np.reciprocal}(Q) \]
\[ \text{exret} = \Pi \]
print(f"Ex post returns to purchase of Arrow securities = \{exret\}\)"
\[ \text{exant} = \Pi \times P \]

\[
\text{print}(f"\text{Ex ante returns to purchase of Arrow securities (exant)}")
\]

\[
P = \begin{bmatrix} 0.8 & 0.2 \\ 0.4 & 0.6 \end{bmatrix}
\]

\[
Q = \begin{bmatrix} 0.768 & 0.192 \\ 0.384 & 0.576 \end{bmatrix}
\]

Govt expenditures in peace and war = [1, 2]

Constant tax collections = 1.3116883116883118

Govt assets in two states = [0, 1.62337662]

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 war security in peace = 0.3116883116883117

Spending on Arrow war security in war = 0.9350649350649349

Government tax collections plus asset levels in peace and war

\[
T+b \text{ in peace} = 1.3116883116883118 \\
T+b \text{ in war} = 2.9350649350649354
\]

Total government spending in peace and war

\[
\text{Peace} = 1.3116883116883118 \\
\text{War} = 2.935064935064935
\]

Let's see ex post and ex ante returns on Arrow securities

\[
\text{Ex post returns to purchase of Arrow securities} = \begin{bmatrix} 1.30208333 & 5.20833333 \\ 2.60416667 & 1.73611111 \end{bmatrix}
\]

\[
\text{Ex ante returns to purchase of Arrow securities} = \begin{bmatrix} 1.04166667 & 1.04166667 \\ 1.04166667 & 1.04166667 \end{bmatrix}
\]

\[\text{Explanation}\]

In this example, the government always purchase 0 units of the Arrow security that pays off in peace time (Markov state 1)

But it purchases a positive amount of the security that pays off in war time (Markov state 2)

We recommend plugging the quantities computed above into the government budget constraints in the two Markov states and staring

This is an example in which the government purchases insurance against the possibility that war breaks out or continues

• the insurance does not pay off so long as peace continues
• the insurance pays off when there is war

**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 \(b_2 = -10\)

### 6.14.6 Linear State Space Version of Complete Markets Model

Now we use a setting like that in the **first lecture on the permanent income model**

In that model, there were

- incomplete markets: the consumer could trade only a single risk-free one-period bond bearing gross one-period risk-free interest rate equal to \(\beta^{-1}\)
- the consumers exogenous nonfinancial income was 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**

Well write down a complete markets counterpart of that model

So now we suppose that nonfinancial income is governed by the state space system

\[
x_{t+1} = Ax_t + Cw_{t+1} \\
y_t = S_y x_t
\]

where \(x_t\) is an \(n \times 1\) vector and \(w_{t+1} \sim N(0, I)\) is IID over time

Again, as a counterpart of the Hall-Barro assumption that the risk-free gross interest rate is \(\beta^{-1}\), we assume the scaled prices of one-period ahead Arrow securities are

\[
p_{t+1}(x_{t+1} \mid x_t) = \beta \phi(x_{t+1} \mid Ax_t, CC')
\]

where \(\phi(\cdot \mid \mu, \Sigma)\) is a multivariate Gaussian distribution with mean vector \(\mu\) and covariance matrix \(\Sigma\)

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 function assumed in (6.139), the value at \(t\) of \(b(x_{t+1})\) is

\[
\beta \int b(x_{t+1}) \phi(x_{t+1} \mid Ax_t, CC') dx_{t+1} = \beta \mathbb{E}_t b_{t+1}
\]

In the complete markets setting, the consumer faces a sequence of budget constraints

\[
c_t + b_t = y_t + \beta \mathbb{E}_t b_{t+1}, t \geq 0
\]

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, we can assume a more general form of utility function that satisfies \( u' > 0 \) and \( u'' < 0 \)
The first-order condition for the consumers problem with complete markets and our assumption about Arrow securities prices is
\[
u'(c_{t+1}) = u'(c_t) \quad \text{for all } t \geq 0
\]
which again implies \( c_t = \bar{c} \) for some \( \bar{c} \)
So it follows that
\[
b_t = E_t \sum_{j=0}^{\infty} \beta^j (y_{t+j} - \bar{c})
\]
or
\[
b_t = S_y (I - \beta A)^{-1} x_t - \frac{1}{1 - \beta} \bar{c} \tag{6.140}
\]
where the value of \( \bar{c} \) satisfies
\[
\bar{b}_0 = S_y (I - \beta A)^{-1} x_0 - \frac{1}{1 - \beta} \bar{c} \tag{6.141}
\]
where \( \bar{b}_0 \) is an initial level of the consumers debt, specified as a parameter of the problem
Thus, in the complete markets version of the consumption-smoothing model, \( c_t = \bar{c}, \forall t \geq 0 \) is determined by (6.141) and the consumers debt is a fixed function of the state \( x_t \) described by (6.140)
Heres 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.

```python
def complete_ss(\beta, 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 \((I - \beta A)^{-1}\)
rm = la.inv(np.eye(A.shape[0]) - np.zeros((1, 1))

# Constant level of consumption
cbar = (1 - \beta) * (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 - \beta))

return c_hist, b_hist, np.squeeze(y_hist), x_hist

# Define parameters
N_simul = 150
\(\alpha, \rho_1, \rho_2 = 10.0, 0.9, 0.0\)
\(\sigma = 1.0\)

A = np.array([[1., 0., 0.],
              [\alpha, \rho_1, \rho_2],
              [0., 1., 0.]])
C = np.array([[0.], [\sigma], [0.]])
S_y = np.array([[1, 1.0, 0.0]])
\(\beta, b0 = 0.95, -10.0\)
x0 = np.array([1.0, \alpha / (1 - \rho_1), \alpha / (1 - \rho_1)])

# 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(\(\beta, b0, x0, A, C, S_y, 150\))
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('Cons and income', fontsize=17)
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='')
Interpretation of Graph

In the above graph, please note that:

- nonfinancial income fluctuates in a stationary manner
- consumption is completely constant
- the consumers debt fluctuates in a stationary manner; in fact, in this case because nonfinancial income is a first-order autoregressive process, the consumers debt is an exact affine function (meaning linear plus a constant) of the consumers nonfinancial income

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 version, consumption follows a random walk and the consumers debt follows a process with a unit root.
We leave it to the reader to apply the usual isomorphism to deduce the corresponding implications for a tax-smoothing model like Barros [Bar79]

**Government Manipulation of Arrow Securities Prices**

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.

In *optimal taxation with incomplete markets*, we study an incomplete-markets model in which the government manipulates asset prices.

### 6.15 Optimal Savings III: Occasionally Binding Constraints

**Contents**

- Optimal Savings III: Occasionally Binding Constraints
  - Overview
  - The Optimal Savings Problem
  - Computation
  - Exercises
  - Solutions

**6.15.1 Overview**

Next we study an optimal savings problem for an infinitely lived consumer—the common ancestor described in [LS18], section 1.3.

This is an essential sub-problem for many representative macroeconomic models

- [Aiy94]
- [Hug93]
- etc.

It is related to the decision problem in the stochastic optimal growth model and yet differs in important ways. For example, the choice problem for the agent includes an additive income term that leads to an occasionally binding constraint.

Our presentation of the model will be relatively brief

- For further details on economic intuition, implication and models, see [LS18]
- 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)
- More efficient numerically than value function iteration

**References**

Other useful references include [Dea91], [DH10], [Kuh13], [Rab02], [Rei09] and [SE77]

### 6.15.2 The Optimal Savings Problem

Let’s write down the model and then discuss how to solve it.

**Set Up**

Consider a household that chooses a state-contingent consumption plan \( \{c_t\}_{t \geq 0} \) to maximize

\[
E \sum_{t=0}^{\infty} \beta^t u(c_t)
\]

subject to

\[
c_t + a_{t+1} \leq Ra_t + z_t, \quad c_t \geq 0, \quad a_t \geq -b \quad t = 0, 1, \ldots
\]

(6.142)

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 \( \Pi \)

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

\[
\int f(\tilde{z}) \Pi(z, d\tilde{z})
\]

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 (6.142), and
3. measurability of \(c_t\) w.r.t. the filtration generated by \(\{z_1, \ldots, z_t\}\)

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

**Value Function and Euler Equation**

The value function \(V : S \to \mathbb{R}\) is defined by

\[
V(a, z) := \sup \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\}
\]  

(6.143)

where the supremum is over all 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 (6.143)

To pin down such paths we can use a version of the Euler equation, which in the present setting is

\[
u'(c_t) \geq \beta R \mathbb{E}_t [u'(c_{t+1})]
\]  

(6.144)

and

\[
u'(c_t) = \beta R \mathbb{E}_t [u'(c_{t+1})] \text{ whenever } c_t < Ra_t + z_t + b
\]  

(6.145)

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 (6.144) and (6.145) are equivalent to

\[
u'(c_t) = \max \{ \beta R \mathbb{E}_t [u'(c_{t+1})], u'(Ra_t + z_t + b) \}
\]  

(6.146)
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 \((6.146)\) and the transversality condition

\[
\lim_{t \to \infty} \beta^t \mathbb{E} [u'(c_t) a_{t+1}] = 0. \tag{6.147}
\]

Moreover, there exists an optimal consumption function \(c^*: S \to [0, \infty)\) such that the path from \((a, z)\) generated by

\[
(a_0, z_0) = (a, z), \quad z_{t+1} \sim \Pi(z_t, d\gamma), \quad c_t = c^*(a_t, z_t) \quad \text{and} \quad a_{t+1} = Ra_t + z_t - c_t
\]

satisfies both \((6.146)\) and \((6.147)\), and hence is the unique optimal path from \((a, z)\)

In summary, to solve the optimization problem, we need to compute \(c^*\)

6.15.3 Computation

There are two standard ways to solve for \(c^*\)

1. Time iteration (TI) using the Euler equality
2. Value function iteration (VFI)

Let's look at these in turn

Time Iteration

We can rewrite \((6.146)\) to make it a statement about functions rather than random variables

In particular, consider the functional equation

\[
u' \circ c(a, z) = \max \left\{ \gamma \int u' \circ c \left\{ Ra + z - c(a, z), \ \hat{z} \right\} \Pi(z, d\hat{z}), \ u'(Ra + z + b) \right\} \tag{6.148}\]

where \(\gamma := \beta R\) and \(u' \circ c(s) := u'(c(s))\)

Equation \((6.148)\) is a functional equation in \(c\)

In order to identify a solution, let \(\mathcal{C}\) be the set of candidate consumption functions \(c: S \to \mathbb{R}\) such that

- each \(c \in \mathcal{C}\) is continuous and (weakly) increasing
- \(\min Z \leq c(a, z) \leq Ra + z + b\) for all \((a, z) \in S\)

In addition, let \(K: \mathcal{C} \to \mathcal{C}\) be defined as follows:

For given \(c \in \mathcal{C}\), the value \(Kc(a, z)\) is the unique \(t \in J(a, z)\) that solves
\[ u'(t) = \max \left \{ \gamma \int u' \circ c \{ Ra + z - t, \, z \} \Pi(z, d\dot{z}), \, u'(Ra + z + b) \right \} \] (6.149)

where

\[ J(a, z) := \{ t \in \mathbb{R} : \min Z \leq t \leq Ra + z + b \} \] (6.150)

We refer to \( K \) as Colemans policy function operator \([Col90]\)

It is known that

- \( K \) is a contraction mapping on \( \mathcal{C} \) under the metric
  \[ \rho(c, d) := \| u' \circ c - u' \circ d \| := \sup_{s \in S} | u'(c(s)) - u'(d(s)) | \quad (c, d \in \mathcal{C}) \]
- The metric \( \rho \) is complete on \( \mathcal{C} \)
- Convergence in \( \rho \) implies uniform convergence on compacts

In consequence, \( K \) has a unique fixed point \( c^* \in \mathcal{C} \) and \( K^n c \rightarrow c^* \) as \( n \rightarrow \infty \) for any \( c \in \mathcal{C} \)

By the definition of \( K \), the fixed points of \( K \) in \( \mathcal{C} \) coincide with the solutions to (6.148) 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 \( c^* \) is the unique optimal path from \((a_0, z_0) \in S\)

**TL;DR** The unique optimal policy can be computed by picking any \( c \in \mathcal{C} \) and iterating with the operator \( K \) defined in (6.149)

**Value Function Iteration**

The Bellman operator for this problem is given by

\[ T v(a, z) = \max_{0 \leq c \leq Ra + z + b} \left \{ u(c) + \beta \int v(Ra + z - c, \, \dot{z}) \Pi(z, d\dot{z}) \right \} \] (6.151)

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 continous 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
Implementation

Here is the code for a class called `ConsumerProblem` that stores primitives, as well as

- a `bellman_operator` function, which implements the Bellman operator $T$ specified above
- a `coleman_operator` function, which implements the Coleman operator $K$ specified above
- an `initialize` function, which generates suitable initial conditions for iteration

```python
import numpy as np
from scipy.optimize import fminbound, brentq

class ConsumerProblem:
    ""
    A class that stores primitives for the income fluctuation problem. The income process is assumed to be a finite state Markov chain.
    
    Parameters
    ----------
    r : scalar(float), optional(default=0.01)
        A strictly positive scalar giving the interest rate
    β : scalar(float), optional(default=0.96)
        The discount factor, must satisfy $(1 + r) \ast \beta < 1$
    Π : array_like(float), optional(default=((0.60, 0.40), (0.05, 0.95)))
        A 2D NumPy array giving the Markov matrix for $\{z_t\}$
    z_vals : array_like(float), optional(default=(0.5, 0.95))
        The state space of $\{z_t\}$
    b : scalar(float), optional(default=0)
        The borrowing constraint
    grid_max : scalar(float), optional(default=16)
        Max of the grid used to solve the problem
    grid_size : scalar(int), optional(default=50)
        Number of grid points to solve problem, a grid on $[-b, grid_max]$
    u : callable, optional(default=np.log)
        The utility function
    du : callable, optional(default=lambda x: 1/x)
        The derivative of $u$
    
    Attributes
    ----------
    r, β, Π, z_vals, b, u, du : see Parameters
    asset_grid : np.ndarray
        One dimensional grid for assets
    ""

    def __init__(self,
        r=0.01,
        β=0.96,
        Π=((0.6, 0.4), (0.05, 0.95)),
        z_vals=(0.5, 1.0),
        b=0,
        grid_max=16,
```
grid_size=50,
u=np.log,
du=\lambda x: 1/x):

self.u, self.du = u, du
self.r, self.R = r, 1 + r
self.\beta, self.b = \beta, b
self.\Pi, self.z_vals = np.array(\Pi), tuple(z_vals)
self.asset_grid = np.linspace(-b, grid_max, grid_size)

def bellman_operator(V, cp, return_policy=False):
    
    The approximate Bellman operator, which computes and returns the
    updated value function TV (or the V-greedy policy c if
    return_policy is True).

    Parameters
    ----------
    V : array_like(float)
        A NumPy array of dim len(cp.asset_grid) times len(cp.z_vals)
    cp : ConsumerProblem
        An instance of ConsumerProblem that stores primitives
    return_policy : bool, optional(default=False)
        Indicates whether to return the greed policy given V or the
        updated value function TV. Default is TV.

    Returns
    -------
    array_like(float)
        Returns either the greed policy given V or the updated value
        function TV.

    ""
    
    # === Simplify names, set up arrays === #
    R, \Pi, \beta, u, b = cp.R, cp.\Pi, cp.\beta, cp.u, cp.b
    asset_grid, z_vals = cp.asset_grid, cp.z_vals
    new_V = np.empty(V.shape)
    new_c = np.empty(V.shape)
    z_idx = list(range(len(z_vals)))

    # === Linear interpolation of V along the asset grid === #
    vf = \lambda a, i_z: np.interp(a, asset_grid, V[:, i_z])

    # === Solve r.h.s. of Bellman equation ===#
    for i_a, a in enumerate(asset_grid):
        for i_z, z in enumerate(z_vals):
            def obj(c):
                # objective function to be *minimized*
                y = sum(vf(R * a + z - c, j) * \Pi[i_z, j] for j in z_idx)
                return -u(c) - \beta * y
            c_star = fminbound(obj, 1e-8, R * a + z + b)
            new_c[i_a, i_z], new_V[i_a, i_z] = c_star, -obj(c_star)
if return_policy:
    return new_c
else:
    return new_V

def coleman_operator(c, cp):
    """
    The approximate Coleman operator.
    
    Iteration with this operator corresponds to time iteration on the Euler
equation. Computes and returns the updated consumption policy
c. The array c is replaced with a function cf that implements
univariate linear interpolation over the asset grid for each
possible value of z.

    Parameters
    ----------
    c : array_like(float)
        A NumPy array of dim len(cp.asset_grid) times len(cp.z_vals)
cp : ConsumerProblem
        An instance of ConsumerProblem that stores primitives

    Returns
    -------
    array_like(float)
        The updated policy, where updating is by the Coleman
        operator.
    """
    # === simplify names, set up arrays === #
    asset_grid, z_vals = cp.asset_grid, cp.z_vals
    z_size = len(z_vals)
    γ = R * β
    vals = np.empty(z_size)

    # === linear interpolation to get consumption function === #
    def cf(a):
        """
        The call cf(a) returns an array containing the values c(a, z) for each z in z_vals. For each such z, the value c(a, z)
is constructed by univariate linear approximation over asset
space, based on the values in the array c
        """
        for i in range(z_size):
            vals[i] = np.interp(a, asset_grid, c[:, i])
        return vals

    # === solve for root to get Kc === #
    Kc = np.empty(c.shape)
    for i_a, a in enumerate(asset_grid):
        for i_z, z in enumerate(z_vals):
            def h(t):
def initialize(cp):
    
    Creates a suitable initial conditions V and c for value function and time iteration respectively.

    Parameters
    ----------
    cp : ConsumerProblem
        An instance of ConsumerProblem that stores primitives
    
    Returns
    -------
    V : array_like(float)
        Initial condition for value function iteration
    c : array_like(float)
        Initial condition for Coleman operator iteration

    
    # === Simplify names, set up arrays === #
    R, β, u, b = cp.R, cp.β, cp.u, cp.b
    asset_grid, z_vals = cp.asset_grid, cp.z_vals
    shape = len(asset_grid), len(z_vals)
    V, c = np.empty(shape), np.empty(shape)

    # === Populate V and c === #
    for i_a, a in enumerate(asset_grid):
        for i_z, z in enumerate(z_vals):
            c_max = R * a + z + b
            c[i_a, i_z] = c_max
            V[i_a, i_z] = u(c_max) / (1 - β)

    return V, c

Both bellman_operator and coleman_operator use linear interpolation along the asset grid to approximate the value and consumption functions.

The following exercises walk you through several applications where policy functions are computed.

In exercise 1 you will see that while VFI and TI produce similar results, the latter is much faster.

Intuition behind this fact was provided in a previous lecture on time iteration.
6.15.4 Exercises

Exercise 1

The first exercise is to replicate the following figure, which compares TI and VFI as solution methods.

The figure shows consumption policies computed by iteration of $K$ and $T$ respectively.

- In the case of iteration with $T$, the final value function is used to compute the observed policy.

Consumption is shown as a function of assets with income $z$ held fixed at its smallest value.

The following details are needed to replicate the figure:

- The parameters are the default parameters in the definition of consumerProblem.
- The initial conditions are the default ones from initialize().
- Both operators are iterated 80 times.

When you run your code you will observe that iteration with $K$ is faster than iteration with $T$.

In the IPython shell, a comparison of the operators can be made as follows.
\[
\text{cp} = \text{ConsumerProblem()}
\]
\[
v, c = \text{initialize(cp)}
\]

\[
\texttt{\%timeit bellman_operator(v, cp)}
\]
\[
10\text{ loops, best of 3: 142 ms per loop}
\]

\[
\texttt{\%timeit coleman_operator(c, cp)}
\]
\[
10\text{ loops, best of 3: 24.9 ms per loop}
\]

The output shows that Coleman operator is about 6 times faster

From now on we will only use the Coleman operator

**Exercise 2**

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

**Exercise 3**

Now lets consider the long run asset levels held by households

Well 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

```python
import matplotlib.pyplot as plt
import quantecon asqe

# === solve for optimal consumption ===#
```
m = ConsumerProblem(r=0.03, grid_max=4)
v_init, c_init = initialize(m)

K = lambda c: coleman_operator(c, m)
c = qe.compute_fixed_point(K, c_init, verbose=False)
a = m.asset_grid
R, z_vals = m.R, m.z_vals

# === generate savings plot === #
fig, ax = plt.subplots(figsize=(10, 8))
ax.plot(a, R * a + z_vals[0] - c[:, 0], label='Low income')
ax.plot(a, R * a + z_vals[1] - c[:, 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

\[ a' = h(a, z) := Ra + z - c^*(a, z) \]
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 [HP92]
- 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`

### Exercise 4

Following on from exercises 2 and 3, let's look at how savings and aggregate asset holdings vary with the interest rate

• Note: [LS18] section 18.6 can be consulted for more background on the topic treated in this exercise

For a given parameterization of the model, the mean of the stationary distribution 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 4 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.

### 6.15.5 Solutions

**Exercise 1**

```python
cp = ConsumerProblem()
K = 80

# Bellman iteration
V, c = initialize(cp)
print("Starting value function iteration")
for i in range(K):
    # print f"Current iterate = {i}")
    V = bellman_operator(V, cp)
c1 = bellman_operator(V, cp, return_policy=True)

# Policy iteration
print("Starting policy function iteration")
V, c2 = initialize(cp)
for i in range(K):
    # print f"Current iterate = {i}")
    c2 = coleman_operator(c2, cp)

fig, ax = plt.subplots(figsize=(10, 8))
ax.plot(cp.asset_grid, c1[:, 0], label='value function iteration')
ax.plot(cp.asset_grid, c2[:, 0], label='policy function iteration')
ax.set_xlabel('asset level')
ax.set_ylabel('consumption (low income)')
ax.legend(loc='upper left')
plt.show()
```

Starting value function iteration
Starting policy function iteration
Exercise 2

```python
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)
    v_init, c_init = initialize(cp)
    K = lambda c: coleman_operator(c, cp)
    c = qe.compute_fixed_point(K, c_init, verbose=False)
    ax.plot(cp.asset_grid, c[:, 0], label=f'r = {r_val:.3f}')

ax.set_xlabel('asset level')
ax.set_ylabel('consumption (low income)')
ax.legend(loc='upper left')
plt.show()
```
Exercise 3

```python
from quantecon import MarkovChain

def compute_asset_series(cp, T=500000, verbose=False):
    
    """
    Simulates a time series of length T for assets, given optimal savings behavior. Parameter cp is an instance of ConsumerProblem
    """

    \Pi, z_vals, R = cp.\Pi, cp.z_vals, cp.R  # Simplify names
    mc = MarkovChain(\Pi)
    v_init, c_init = initialize(cp)
    K = lambda c: coleman_operator(c, cp)
    c = qe.compute_fixed_point(K, c_init, verbose=verbose)
    cf = lambda a, i_z: np.interp(a, cp.asset_grid, c[:, i_z])
    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)
```

6.15. Optimal Savings III: Occasionally Binding Constraints
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, normed=True)
ax.set_xlabel('assets')
ax.set_xlim(-0.05, 0.75)
plt.show()

Exercise 4

The following code takes a little while to run

\[
M = 25
\]

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_yticks(np.arange(.0, 0.045, .01))
ax.set_xticks(np.arange(-3, 2, 1))
ax.set_xlabel('capital')
ax.set_ylabel('interest rate')
ax.grid(True)
ax.legend(loc='upper left')
plt.show()

Finished iteration b=1
Finished iteration b=3
6.16 Robustness

6.16.1 Overview

This lecture modifies a Bellman equation to express a decision maker's doubts about transition dynamics. His specification doubts make the decision maker want a robust decision rule. Robust means insensitive to misspecification of transition dynamics.

The decision maker has a single approximating model. He calls it approximating to acknowledge that he doesn't completely trust it. He fears that outcomes will actually be determined by another model that he cannot describe explicitly.

All that he knows is that the actual data-generating model is in some (uncountable) set of models that surrounds his approximating model.

He quantifies the discrepancy between his approximating model and the genuine data-generating model by using a quantity called entropy.

(Well 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.

### 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 \textit{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 makers approximating model. Here

- \textit{Value} refers to a sum of discounted rewards obtained by applying the decision rule $F$ when the state starts at some fixed initial state $x_0$
- \textit{Entropy} is a nonnegative number that measures the size of a set of models surrounding the decision makers 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 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 well use to construct these sets

- Well use some min problems to construct the lower bounds
- Well 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)

**Inspiring Video**

If you want to understand more about why one serious quantitative researcher is interested in this approach, we recommend Lars Peter Hansens Nobel lecture

**Other References**

Our discussion in this lecture is based on

- [HS00]
- [HS08]

**6.16.2 The Model**

For simplicity, we present ideas in the context of a class of problems with linear transition laws and quadratic objective functions

To fit in with our earlier lecture on LQ control, we will treat loss minimization rather than value maximization

To begin, recall the infinite horizon LQ problem, where an agent chooses a sequence of controls $\{u_t\}$ to minimize

$$\sum_{t=0}^{\infty} \beta^t \left\{ x_t' R x_t + u_t' Q u_t \right\}$$  \hspace{1cm} (6.152)

subject to the linear law of motion
\[ x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \quad t = 0, 1, 2, \ldots \] (6.153)

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 will quantify the quality of a model specification in terms of the maximal size of the expression

\[ \sum_{t=0}^{\infty} \beta^{t+1} w_{t+1}^t w_{t+1} \]

### 6.16.3 Constructing More Robust Policies

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

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

(Here \( J \) depends on \( t \) because the sequence \( \{w_t\} \) is not recursive)

Our tool for studying robustness is to construct a rule that works well even if an adverse sequence \( \{w_t\} \) occurs.

In our framework, adverse means loss increasing.

As well see, this will eventually lead us to construct the Bellman equation

\[ J(x) = \min_u \max_w \{ x'Rx + u'Qu + \beta [J(Ax + Bu + Cw) - \theta w'w] \} \] (6.154)

Notice that weve added the penalty term \(-\theta w'w\).

Since \( w'w = \|w\|^2 \), this term becomes influential when \( w \) moves away from the origin.
The penalty parameter $\theta$ controls how much we penalize the maximizing agent for harming the minimizing agent.

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

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

### Analyzing the Bellman equation

So what does $J$ in (6.154) 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

$$
\max_w \{(Ax + Bu + Cw)'P(Ax + Bu + Cw) - \theta w'w\} = (Ax + Bu)'\mathcal{D}(P)(Ax + Bu)
$$

(6.155)

where

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

(6.156)

and $I$ is a $j \times j$ identity matrix. Substituting this expression for the maximum into (6.154) yields

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

(6.157)

Using similar mathematics, the solution to this minimization problem is $u = -Fx$ where $F := (Q + \beta B'D(P)B)^{-1}\beta B'D(P)A$

Substituting this minimizer back into (6.157) and working through the algebra gives $x'Px = x'B(D(P))x$

for all $x$, or, equivalently,

$$
P = B(D(P))
$$

where $D$ is the operator defined in (6.156) and

$$
B(P) := R - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA + \beta A'PA
$$

The operator $B$ is the standard (i.e., non-robust) LQ Bellman operator, and $P = B(P)$ is the standard matrix Riccati equation coming from the Bellman equation see *this discussion*.
Under some regularity conditions (see [HS08]), the operator $B \circ D$ has a unique positive definite fixed point, which we denote below by $\hat{P}$.

A robust policy, indexed by $\theta$, is $u = -\hat{F}x$ where

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

(6.158)

We also define

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

(6.159)

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 (6.155) evaluated at the fixed rule $u = -\hat{F}x$.

Note that $\hat{P}$, $\hat{F}$, $\hat{K}$ are all determined by the primitives and $\theta$.

Note also that if $\theta$ is very large, then $D$ is approximately equal to the identity mapping.

Hence, when $\theta$ is large, $\hat{P}$ and $\hat{F}$ are approximately equal to their standard LQ values.

Furthermore, when $\theta$ is large, $\hat{K}$ is approximately equal to zero.

Conversely, smaller $\theta$ is associated with greater fear of model misspecification, and greater concern for robustness.

### 6.16.4 Robustness as Outcome of a Two-Person Zero-Sum Game

What we have done above can be interpreted in terms of a two-person zero-sum game in which $\hat{F}$, $\hat{K}$ are Nash equilibrium objects.

Agent 1 is our original agent, who seeks to minimize loss in the LQ program while admitting the possibility of misspecification.

Agent 2 is an imaginary malevolent player.

Agent 2’s malevolence helps the original agent to compute bounds on his value function across a set of models.

We begin with agent 2’s problem.

**Agent 2’s Problem**

1. knows a fixed policy $F$ specifying the behavior of agent 1, in the sense that $u_t = -F x_t$ for all $t$.
2. responds by choosing a shock sequence $\{w_t\}$ from a set of paths sufficiently close to the benchmark sequence $\{0, 0, 0, \ldots\}$.
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

\[ \sum_{t=1}^{\infty} \beta^t w'_t w_t \leq \eta \] (6.160)

Now let \( F \) be a fixed policy, and let \( J_F(x_0, w) \) be the present-value cost of that policy given sequence \( w := \{w_t\} \) and initial condition \( x_0 \in \mathbb{R}^n \)

Substituting \(-Fx_t\) for \( u_t \) in (6.152), this value can be written as

\[ J_F(x_0, w) := \sum_{t=0}^{\infty} \beta^t x'_t (R + F'QF)x_t \] (6.161)

where

\[ x_{t+1} = (A - BF)x_t + Cw_{t+1} \] (6.162)

and the initial condition \( x_0 \) is as specified in the left side of (6.161)

Agent 2 chooses \( w \) to maximize agent 1s loss \( J_F(x_0, w) \) subject to (6.160)

Using a Lagrangian formulation, we can express this problem as

\[ \max_w \sum_{t=0}^{\infty} \beta^t \left( x'_t (R + F'QF)x_t - \beta \theta (w'_{t+1} w_{t+1} - \eta) \right) \]

where \( \{x_t\} \) satisfied (6.162) and \( \theta \) is a Lagrange multiplier on constraint (6.160)

For the moment, lets take \( \theta \) as fixed, allowing us to drop the constant \( \beta \theta \eta \) term in the objective function, and hence write the problem as

\[ \max_w \sum_{t=0}^{\infty} \beta^t \left( x'_t (R + F'QF)x_t - \beta w'_{t+1} w_{t+1} \right) \]

or, equivalently,

\[ \min_w \sum_{t=0}^{\infty} \beta^t \left( -x'_t (R + F'QF)x_t + \beta w'_{t+1} w_{t+1} \right) \] (6.163)

subject to (6.162)

Whats striking about this optimization problem is that it is once again an LQ discounted dynamic programming problem, with \( w = \{w_t\} \) as the sequence of controls

6.16. Robustness
The expression for the optimal policy can be found by applying the usual LQ formula (see here). We denote it by $K(F, \theta)$, with the interpretation $w_{t+1} = K(F, \theta)x_t$.

The remaining step for agent 2's problem is to set $\theta$ to enforce the constraint (6.160), which can be done by choosing $\theta = \theta_\eta$ such that

$$
\beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \theta_\eta)'K(F, \theta_\eta)x_t = \eta 
$$

(6.164)

Here $x_t$ is given by (6.162) which in this case becomes $x_{t+1} = (A - BF + CK(F, \theta))x_t$

**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 (6.163) as $R_\theta(x_0, F)$.

Because minimizers minimize we have

$$
R_\theta(x_0, F) \leq \sum_{t=0}^{\infty} \beta^t \{ -x_t'(R + F'QF)x_t \} + \beta \sum_{t=0}^{\infty} \beta^t w_{t+1}'w_{t+1},
$$

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

$$
R_\theta(x_0, F) - \theta \text{ ent} \leq \sum_{t=0}^{\infty} \beta^t \{ -x_t'(R + F'QF)x_t \}
$$

(6.165)

where

$$
\text{ent} := \beta \sum_{t=0}^{\infty} \beta^t w_{t+1}'w_{t+1}
$$

The left side of inequality (6.165) 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 (6.165) is attained when

$$
\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x_t'K(F, \theta)'K(F, \theta)x_t
$$

(6.166)

To construct the lower bound on the set of values associated with all perturbations $w$ satisfying the entropy constraint (6.160) at a given entropy level, we proceed as follows:
• For a given $\theta$, solve the minimization problem (6.163)
• Compute the minimizer $R_\theta(x_0, F)$ and the associated entropy using (6.166)
• Compute the lower bound on the value function $R_\theta(x_0, F) - \theta \text{ent}$ and plot it against $\text{ent}$
• Repeat the preceding three steps for a range of values of $\theta$ to trace out the lower bound

**Note:** This procedure sweeps out a set of separating hyperplanes indexed by different values for the Lagrange multiplier $\theta$

### The Upper Bound

To construct an upper bound we use a very similar procedure

We simply replace the minimization problem (6.163) with the maximization problem

$$V_{\tilde{\theta}}(x_0, F) = \max_w \sum_{t=0}^{\infty} \beta^t \left\{ -x'_t (R + F'QF)x_t - \tilde{\theta}w'_{t+1}w_{t+1} \right\}$$

where now $\tilde{\theta} > 0$ penalizes the choice of $w$ with larger entropy.

(Notice that $\tilde{\theta} = -\theta$ in problem (6.163))

Because maximizers maximize we have

$$V_{\tilde{\theta}}(x_0, F) \geq \sum_{t=0}^{\infty} \beta^t \left\{ -x'_t (R + F'QF)x_t \right\} - \tilde{\theta} \sum_{t=0}^{\infty} \beta^t w'_{t+1}w_{t+1}$$

which in turn implies the inequality

$$V_{\tilde{\theta}}(x_0, F) + \tilde{\theta} \text{ent} \geq \sum_{t=0}^{\infty} \beta^t \left\{ -x'_t (R + F'QF)x_t \right\}$$

where

$$\text{ent} \equiv \beta \sum_{t=0}^{\infty} \beta^t w'_{t+1}w_{t+1}$$

The left side of inequality (6.168) is a straight line with slope $\tilde{\theta}$

The upper bound on the left side of (6.168) is attained when

$$\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x'_t K(F, \tilde{\theta})'K(F, \tilde{\theta})x_t$$

To construct the upper bound on the set of values associated all perturbations $w$ with a given entropy we proceed much as we did for the lower bound
• For a given $\tilde{\theta}$, solve the maximization problem (6.167)
• Compute the maximizer $V_\tilde{\theta}(x_0, F)$ and the associated entropy using (6.169)
• 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

**Agent 1’s Problem**

Now we turn to agent 1, who solves

$$
\min_{\{u_t\}} \sum_{t=0}^{\infty} \beta^t \{ x_t'Rx_t + u_t'Qu_t - \beta \theta w_{t+1}'w_{t+1} \}
$$

(6.170)

where $\{w_{t+1}\}$ satisfies $w_{t+1} = Kx_t$

In other words, agent 1 minimizes

$$
\sum_{t=0}^{\infty} \beta^t \{ x_t'(R - \beta \theta K'K)x_t + u_t'Qu_t \}
$$

(6.171)

subject to

$$
x_{t+1} = (A + CK)x_t + Bu_t
$$

(6.172)

Once again, the expression for the optimal policy can be found here we denote it by $\tilde{F}$

**Nash Equilibrium**

Clearly the $\tilde{F}$ we have obtained depends on $K$, which, in agent 2’s problem, depended on an initial policy $F$

Holding all other parameters fixed, we can represent this relationship as a mapping $\Phi$, where

$$
\tilde{F} = \Phi(K(F, \theta))
$$

The map $F \mapsto \Phi(K(F, \theta))$ corresponds to a situation in which

1. agent 1 uses an arbitrary initial policy $F$
2. agent 2 best responds to agent 1 by choosing $K(F, \theta)$
3. agent 1 best responds to agent 2 by choosing \( \tilde{F} = \Phi(K(F, \theta)) \)

As you may have already guessed, the robust policy \( \tilde{F} \) defined in (6.158) is a fixed point of the mapping \( \Phi \).

In particular, for any given \( \theta \),

1. \( K(\tilde{F}, \theta) = \tilde{K} \), where \( \tilde{K} \) is as given in (6.159)
2. \( \Phi(\tilde{K}) = \tilde{F} \)

A sketch of the proof is given in the appendix.

### 6.16.5 The Stochastic Case

Now we turn to the stochastic case, where the sequence \( \{w_t\} \) is treated as an iid sequence of random vectors.

In this setting, we suppose that our agent is uncertain about the conditional probability distribution of \( w_{t+1} \).

The agent takes the standard normal distribution \( N(0, I) \) as the baseline conditional distribution, while admitting the possibility that other nearby distributions prevail.

These alternative conditional distributions of \( w_{t+1} \) might depend nonlinearly on the history \( x_s, s \leq t \).

To implement this idea, we need a notion of what it means for one distribution to be near another one.

Here we adopt a very useful measure of closeness for distributions known as the relative entropy, or Kullback-Leibler divergence.

For densities \( p, q \), the Kullback-Leibler divergence of \( q \) from \( p \) is defined as

\[
D_{KL}(p, q) := \int \ln \left[ \frac{p(x)}{q(x)} \right] p(x) \, dx
\]

Using this notation, we replace (6.154) with the stochastic analogue

\[
J(x) = \min_u \max_{\psi \in \mathcal{P}} \left\{ x'Rx + u'Qu + \beta \left[ \int J(Ax + Bu + Cw) \psi(dw) - \theta D_{KL}(\psi, \phi) \right] \right\}
\]  

(6.173)

Here \( \mathcal{P} \) represents the set of all densities on \( \mathbb{R}^n \) and \( \phi \) is the benchmark distribution \( N(0, I) \).

The distribution \( \phi \) is chosen as the least desirable conditional distribution in terms of next period outcomes, while taking into account the penalty term \( \theta D_{KL}(\psi, \phi) \).

This penalty term plays a role analogous to the one played by the deterministic penalty \( \theta w'w \) in (6.154), since it discourages large deviations from the benchmark.

### Solving the Model

The maximization problem in (6.173) 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.

6.16. Robustness 747
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
\[
\max_{\psi \in \mathcal{P}} \left\{ \int (Ax + Bu + Cw)'P(Ax + Bu + Cw)\psi(dw) - \theta D_{KL}(\psi, \phi) \right\}
= (Ax + Bu)'D(P)(Ax + Bu) + \kappa(\theta, P)
\]
where
\[
\kappa(\theta, P) := \theta \ln[\det(I - \theta^{-1}C'PC)^{-1}]
\]
and the maximizer is the Gaussian distribution
\[
\psi = N((\theta I - C'PC)^{-1}C'P(Ax + Bu), (I - \theta^{-1}C'PC)^{-1})
\]
(6.175)

Substituting the expression for the maximum into Bellman equation (6.173) and using \( J(x) = x'Px + d \) gives
\[
x'Px + d = \min_u \left\{ x'Rx + u'Qu + \beta (Ax + Bu)'D(P)(Ax + Bu) + \beta [d + \kappa(\theta, P)] \right\}
\]
(6.176)

Since constant terms do not affect minimizers, the solution is the same as (6.157), leading to
\[
x'Px + d = x'B(D(P))x + \beta [d + \kappa(\theta, P)]
\]
To solve this Bellman equation, we take \( \hat{P} \) to be the positive definite fixed point of \( B \circ D \).

In addition, we take \( \hat{d} \) as the real number solving \( d = \beta [d + \kappa(\theta, P)] \), which is
\[
\hat{d} := \frac{\beta}{1 - \beta} \kappa(\theta, P)
\]
(6.177)

The robust policy in this stochastic case is the minimizer in (6.176), which is once again \( u = -\hat{F}x \) for \( \hat{F} \) given by (6.158).

Substituting the robust policy into (6.175) 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 (6.159).

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.

**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 (6.173), satisfies

\[
J_F(x) = \max_{\psi \in \mathcal{P}} \left\{ x'(R + F'QF)x + \beta \left[ \int J_F((A - BF)x + Cw) \psi(dw) - \theta D_{KL}(\psi, \phi) \right] \right\}
\]

Writing \( J_F(x) = x'P_Fx + d_F \) and applying the same argument used to derive (6.174) we get

\[
x'P_Fx + d_F = x'(R + F'QF)x + \beta \left[ x'(A - BF)'\mathcal{D}(P_F)(A - BF)x + d_F + \kappa(\theta, P_F) \right]
\]

To solve this we take \( P_F \) to be the fixed point

\[
P_F = R + F'QF + \beta(A - BF)'\mathcal{D}(P_F)(A - BF)
\]

and

\[
d_F := \frac{\beta}{1 - \beta} \kappa(\theta, P_F) = \frac{\beta}{1 - \beta} \theta \ln[\det(I - \theta^{-1}C'P_FC)^{-1}] \tag{6.178}
\]

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.

### 6.16.6 Implementation

The **QuantEcon.py** package provides a class called **RBLQ** for implementation of robust LQ optimal control. The code can be found on GitHub.

Here is a brief description of the methods of the class:

- **d_operator()** and **b_operator()** implement \( \mathcal{D} \) and \( \mathcal{B} \) respectively
- **robust_rule()** and **robust_rule_simple()** both solve for the triple \( \hat{F}, \hat{K}, \hat{P} \), as described in equations (6.158) – (6.159) 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 (6.164)
- **evaluate_F()** computes the loss and entropy associated with a given policy see this discussion
**6.16.7 Application**

Let us consider a monopolist similar to *this one*, but now facing model uncertainty. The inverse demand function is

\[ p_t = a_0 - a_1 y_t + d_t \]

where

\[ d_{t+1} = p d_t + \sigma_d w_{t+1}, \quad \{w_t\} \overset{\text{iid}}{\sim} N(0, 1) \]

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 \( E \sum_{t=0}^{\infty} \beta^t (-r_t) \)

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

\[ A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \rho \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ 0 \\ \sigma_d \end{bmatrix} \]

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 agents baseline, with uncertainty parameterized by \( \theta \).

We compute value-entropy correspondences for two policies

1. The no concern for robustness policy \( F_0 \), which is the ordinary LQ loss minimizer
2. A moderate concern for robustness policy \( F_\theta \), with \( \theta = 0.02 \)

The code for producing the graph shown above, with blue being for the robust policy, is as follows.
import pandas as pd
import numpy as np
from scipy.linalg import eig
import matplotlib.pyplot as plt
import quantecon as qe

# == 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.],
              [0.1]])

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.
```python
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, 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
eigvals, eigvecs = eig(test_matrix)
assert (eigvals >= 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):

6.16. Robustness
Heres 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?

6.16.8 Appendix

We sketch the proof only of the first claim in this section, which is that, for any given $\theta$, $K(\hat{F}, \theta) = \hat{K}$, where $\hat{K}$ is as given in (6.159)

This is the content of the next lemma
**Lemma.** If \( \hat{P} \) is the fixed point of the map \( B \circ D \) and \( \hat{F} \) is the robust policy as given in (6.158), then

\[
K(\hat{F}, \theta) = (\theta I - C'\hat{P}C)^{-1}C'\hat{P}(A - B\hat{F})
\]  

(6.179)

**Proof:** As a first step, observe that when \( F = \hat{F} \), the Bellman equation associated with the LQ problem (6.162) – (6.163) is

\[
\hat{P} = -R - \hat{F}'Q\hat{F} - \beta^2(A - B\hat{F})' \hat{P}C(\beta\theta I + \beta C'\hat{P}C)^{-1}C'\hat{P}(A - B\hat{F}) + \beta(A - B\hat{F})' \hat{P}(A - B\hat{F})
\]

(6.180)

(revisit this discussion if you don’t know where (6.180) comes from) and the optimal policy is

\[
w_{t+1} = -\beta(\beta\theta I + \beta C'\hat{P}C)^{-1}C'\hat{P}(A - B\hat{F})x_t
\]

Suppose for a moment that \(-\hat{P}\) solves the Bellman equation (6.180)

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 (6.179)

Hence it remains only to show that \(-\hat{P}\) solves (6.180), 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 \(D\), we can rewrite the right-hand side more simply as

\[
R + \hat{F}'Q\hat{F} + \beta(A - B\hat{F})' D(\hat{P})(A - B\hat{F})
\]

Although it involves a substantial amount of algebra, it can be shown that the latter is just \(\hat{P}\)

(Hint: Use the fact that \(\hat{P} = B(D(\hat{P}))\))

### 6.17 Discrete State Dynamic Programming

**Contents**

- *Discrete State Dynamic Programming*
  - Overview
  - Discrete DPs
  - Solving Discrete DPs
  - Example: A Growth Model
6.17.1 Overview

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

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

We call such problems discrete dynamic programs, or discrete DPs

Discrete DPs are the workhorses in much of modern quantitative economics, including
- monetary economics
- search and labor economics
- household savings and consumption theory
- investment theory
- asset pricing
- industrial organization, etc.

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

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

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
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.

References

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

- [LS18]
- [HLL96], section 3.5
- [Put05]
- [SLP89]
- [Rus96]
- [MF02]
- EDTC, chapter 5

6.17.2 Discrete DPs

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

\[ E \sum_{t=0}^{\infty} \beta^t r(s_t, a_t) \]  \hspace{1cm} (6.181)

where

- \( s_t \) is the state variable
- \( a_t \) is the action
- \( \beta \) is a discount factor
- \( r(s_t, a_t) \) is interpreted as a current reward when the state is \( s_t \) and the action chosen is \( a_t \)
Each pair \((s_t, a_t)\) pins down transition probabilities \(Q(s_t, a_t, s_{t+1})\) for the next period state \(s_{t+1}\)

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

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

Examples:

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

**Policies**

The most fruitful way to think about solutions to discrete DP problems is to compare \emph{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 \emph{stationary Markov policies}, which consider only the current state.

In particular, a stationary Markov policy is a map \(\sigma\) from states to actions

\[ a_t = \sigma(s_t) \text{ indicates that } a_t \text{ is the action to be taken in state } s_t \]

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

- See section 5.5 of [Put05] for discussion and proofs.

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

The aim is to find an optimal policy, in the sense of one that maximizes \((6.181)\).

Let's now step through these ideas more carefully.

**Formal definition**

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

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

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

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

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

A policy is a function $\sigma: S \to A$.

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

Denote the set of all feasible policies by $\Sigma$.

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

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

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

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

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

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

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

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

(6.182)

Comments

- $\{s_t\} \sim Q_\sigma$ means that the state is generated by stochastic matrix $Q_\sigma$.
- See this discussion on computing expectations of Markov chains for an explanation of the expression in (6.182).

Notice that were 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.

**Value and Optimality**

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

To calculate this quantity we pass the expectation through the sum in (6.181) and use (6.182) to get

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

This function is called the policy value function for the policy $\sigma$.

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

$$v^*(s) = \max_{\sigma \in \Sigma} v_\sigma(s) \quad (s \in S)$$
(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 $w: S \to \mathbb{R}$, a policy $\sigma \in \Sigma$ is called $w$-greedy if
$$\sigma(s) \in \arg \max_{a \in A(s)} \left\{ r(s, a) + \beta \sum_{s' \in S} w(s') Q(s, a, s') \right\} \quad (s \in S)$$
As discussed in detail below, optimal policies are precisely those that are $v^*$-greedy

Two Operators

It is useful to define the following operators:

- The Bellman operator $T: \mathbb{R}^S \to \mathbb{R}^S$ is defined by
  $$ (Tv)(s) = \max_{a \in A(s)} \left\{ r(s, a) + \beta \sum_{s' \in S} v(s') Q(s, a, s') \right\} \quad (s \in S) $$

- For any policy function $\sigma \in \Sigma$, the operator $T_\sigma: \mathbb{R}^S \to \mathbb{R}^S$ is defined by
  $$ (T_\sigma v)(s) = r(s, \sigma(s)) + \beta \sum_{s' \in S} v(s') Q(s, \sigma(s), s') \quad (s \in S) $$

This can be written more succinctly in operator notation as
$$ T_\sigma v = r_\sigma + \beta Q_\sigma v $$

The two operators are both monotone
- $v \leq w$ implies $Tv \leq Tw$ pointwise on $S$, and similarly for $T_\sigma$

They are also contraction mappings with modulus $\beta$
- $\|Tv - Tw\| \leq \beta \|v - w\|$ and similarly for $T_\sigma$, where $\|\cdot\|$ is the max norm

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

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

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

  or in other words, $v^*$ is the unique fixed point of $T$, and
• \( \sigma^* \) is an optimal policy function if and only if it is \( v^* \)-greedy

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

\[
\sigma^*(s) \in \arg\max_{a \in A(s)} \left\{ r(s, a) + \beta \sum_{s' \in S} v^*(s') Q(s, \sigma(s), s') \right\} \quad (s \in S)
\]

### 6.17.3 Solving Discrete DPs

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

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.

#### Value Function Iteration

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

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

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

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

#### Policy Function Iteration

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

Each iteration consists of

1. A policy evaluation step that computes the value \( v_\sigma \) of a policy \( \sigma \) by solving the linear equation
   \[
   v = T_\sigma v
   \]
2. A policy improvement step that computes a \( v_\sigma \)-greedy policy

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

- See theorem 10.2.6 of \textit{EDTC} for a proof

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

---

6.17. Discrete State Dynamic Programming 761
Modified Policy Function Iteration

Modified policy iteration replaces the policy evaluation step in policy iteration with partial policy evaluation. The latter computes an approximation to the value of a policy $\sigma$ by iterating $T_\sigma$ for a specified number of times. This approach can be useful when the state space is very large and the linear system in the policy evaluation step of policy iteration is correspondingly difficult to solve. The details of the algorithm can be found in the appendix.

6.17.4 Example: A Growth Model

Let’s consider a simple consumption-saving model. A single household either consumes or stores its own output of a single consumption good. The household starts each period with current stock $s$. Next, the household chooses a quantity $a$ to store and consumes $c = s - a$.

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

Output is drawn from a discrete uniform distribution on $\{0, \ldots, B\}$. The next period stock is therefore

$$s' = a + U$$

where $U \sim U[0, \ldots, B]$.

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

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 $S = \{0, \ldots, M + B\}$.
  - hence $n = M + B + 1$.
- the action to be the storage quantity $a$.
- the set of feasible actions at $s$ to be $A(s) = \{0, \ldots, \min\{s, M\}\}$.
  - hence $A = \{0, \ldots, M\}$ and $m = M + 1$.
- the reward function to be $r(s, a) = u(s - a)$.
- the transition probabilities to be...
\[ Q(s, a, s') := \begin{cases} \frac{1}{B+1} & \text{if } a \leq s' \leq a + B \\ 0 & \text{otherwise} \end{cases} \] \quad (6.183)

**Defining a DiscreteDP Instance**

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

1. An \( n \times m \) reward array \( R \)
2. An \( n \times m \times n \) transition probability array \( Q \)
3. A discount factor \( \beta \)

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

For \( Q \) we follow the rule in (6.183)

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:

```python
import numpy as np

class SimpleOG:
    def __init__(self, B=10, M=5, α=0.5, β=0.9):
        """
        Set up \( R, Q \) and \( \beta \), 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 \( R[s, a] = -\infty \) for infeasible
        state-action pairs.
        """

        for s in range(self.n):
            for a in range(self.m):
```

6.17. Discrete State Dynamic Programming
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)

Lets run this code and create an instance of SimpleOG

g = SimpleOG()  # Use default parameters

Instances of DiscreteDP are created using the signature DiscreteDP(R, Q, β)
Lets create an instance using the objects stored in g

import quantecon as qe
ddp = qe.markov.DiscreteDP(g.R, g.Q, g.β)

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

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

Lets see what we've got here

dir(results)

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

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

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

results.v

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

results.sigma
array([0, 0, 0, 0, 1, 1, 1, 2, 2, 3, 3, 4, 5, 5, 5, 5])

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

Let's make sure this didn't happen:

```python
results.max_iter
```

`250`

```python
results.num_iter
```

`3`

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

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

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

```python
results.mc.stationary_distributions
```

```python
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:

---

6.17. Discrete State Dynamic Programming 765
What happens if the agent is more patient?

```python
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
```

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

![Bar graph showing stationary distributions](image)

**State-Action Pair Formulation**

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

One of the advantages of this alternative set up is that it permits 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 \( L \times n \) transition probability array
Heres how we could set up these objects for the preceding example

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

def u(c):
    return c**α

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

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

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

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

### 6.17.5 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

### 6.17.6 Solutions

Written jointly with Diasuke Oyama

**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 \( \beta = 0.95 \)

\[
\begin{align*}
\alpha &= 0.65 \\
f &= \text{lambda } k: k^{\alpha} \\
u &= \text{np.log} \\
\beta &= 0.95
\end{align*}
\]

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 \( \text{grid}_\text{size}=500 \), from \( 10^{-6} \) to \( \text{grid}_\text{max}=2 \)

\[
\begin{align*}
\text{grid}_\text{max} &= 2 \\
\text{grid}_\text{size} &= 500 \\
\text{grid} &= \text{np.linspace}(1e-6, \text{grid}_\text{max}, \text{grid}_\text{size})
\end{align*}
\]

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 \( 0, \ldots, \text{grid}_\text{size}-1 \)

Action (indexed by) \( a \) is feasible at state (indexed by) \( s \) if and only if \( \text{grid}[a] < f([\text{grid}[s]]) \) (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:

\[
\begin{align*}
\text{C} &= f(\text{grid}).\text{reshape}(\text{grid}_\text{size}, 1) - \text{grid}.\text{reshape}(1, \text{grid}_\text{size}) \\
\text{s}_\text{indices}, \text{a}_\text{indices} &= \text{np.where}(\text{C} > 0) \\
\text{L} &= \text{len}(\text{s}_\text{indices})
\end{align*}
\]

Reward vector \( R \) (of length \( \text{L} \)):
\( R = u(C[s\_indices, a\_indices]) \)

(Degenerate) transition probability matrix \( Q \) (of shape \( (L, \text{grid\_size}) \)), where we choose the \( \text{scipy.sparse.lil\_matrix} \) format, while any format will do (internally it will be converted to the csr format):

\[
Q = \text{scipy.sparse.lil\_matrix}((L, \text{grid\_size}))
Q[\text{np.arange}(L), a\_indices] = 1
\]

(If you are familiar with the data structure of \( \text{scipy.sparse.csr\_matrix} \), the following is the most efficient way to create the \( Q \) matrix in the current case)

\[
\# \text{data} = \text{np.ones}(L) \\
\# \text{indptr} = \text{np.arange}(L+1) \\
\# Q = \text{scipy.sparse.csr\_matrix}((\text{data, a\_indices, indptr}), \text{shape}=(L, \text{grid\_size}))
\]

Discrete growth model:

\[ \text{ddp} = \text{DiscreteDP}(R, Q, \beta, s\_indices, a\_indices) \]

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

Solving the model

Solve the dynamic optimization problem:

\[
\text{res} = \text{ddp.solve}(\text{method}='\text{policy\_iteration}') \\
\text{v, } \sigma, \text{num\_iter} = \text{res.v, res.sigma, res.num\_iter} \\
\text{num\_iter} = 10
\]

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

\[
\# \text{Optimal consumption in the discrete version} \\
c = f(\text{grid}) - \text{grid[}\sigma]\]
\[
\# \text{Exact solution of the continuous version} \\
ab = \alpha + \beta \\
c1 = (\text{np.log}(1 - ab) + \text{np.log}(ab) \ast ab / (1 - ab)) / (1 - \beta) \\
c2 = \alpha / (1 - ab) \\
\textbf{def } v\_\text{star}(k): \\
\quad \textbf{return } c1 + c2 \ast \text{np.log}(k)
\]
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

```python
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:

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

`121.49819147053378`

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

`0.012681735127500815`

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

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

0.0038265231000100819

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

False

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

174

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

0.0019618533397668392

The value function is monotone:

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

True

### Comparison of the solution methods

Let us solve the problem by the other two methods

#### Value iteration

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

294

```python
np.array_equal(sigma, res1.sigma)
```

True
Modified policy iteration

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

```
16
```

```python
np.array_equal(\sigma, res2.sigma)
```

```
True
```

Speed comparison

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

```
1 loop, best of 3: 236 ms per loop
10 loops, best of 3: 18.7 ms per loop
10 loops, best of 3: 22.2 ms per loop
```

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

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`

```python
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')
```
We next plot the consumption policies along the value iteration.

```python
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)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Distance</th>
<th>Elapsed (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.518e+00</td>
<td>1.464e-03</td>
</tr>
<tr>
<td>2</td>
<td>4.070e+00</td>
<td>2.330e-03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Distance</th>
<th>Elapsed (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.518e+00</td>
<td>1.046e-03</td>
</tr>
<tr>
<td>2</td>
<td>4.070e+00</td>
<td>1.970e-03</td>
</tr>
<tr>
<td>3</td>
<td>3.866e+00</td>
<td>2.848e-03</td>
</tr>
<tr>
<td>4</td>
<td>3.673e+00</td>
<td>3.715e-03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Distance</th>
<th>Elapsed (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.518e+00</td>
<td>1.052e-03</td>
</tr>
<tr>
<td>2</td>
<td>4.070e+00</td>
<td>1.925e-03</td>
</tr>
<tr>
<td>3</td>
<td>3.866e+00</td>
<td>2.804e-03</td>
</tr>
<tr>
<td>4</td>
<td>3.673e+00</td>
<td>3.654e-03</td>
</tr>
<tr>
<td>5</td>
<td>3.489e+00</td>
<td>4.510e-03</td>
</tr>
<tr>
<td>6</td>
<td>3.315e+00</td>
<td>5.350e-03</td>
</tr>
</tbody>
</table>
6.17. Discrete State Dynamic Programming

The diagrams illustrate the convergence of value function iterations for approximate and true optimal consumption policies.

- **2 value function iterations**
  - Approximate optimal consumption policy
  - True optimal consumption policy

- **4 value function iterations**
  - Approximate optimal consumption policy
  - True optimal consumption policy

- **6 value function iterations**
  - Approximate optimal consumption policy
  - True optimal consumption policy
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 $k_0 = 0.1$

```python
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(0.1, 0.3)

# 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'β = {beta}')

ax.legend(loc='lower right')
plt.show()```
6.17.7 Appendix: Algorithms

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

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

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

**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 $v^{i+1} = T v^i$
3. If $\|v^{i+1} - v^i\| < [(1 - \beta)/(2\beta)]\varepsilon$, then go to step 4; otherwise, set $i = i + 1$ and go to step 2
4. Compute a $v^{i+1}$-greedy policy $\sigma$, and return $v^{i+1}$ and $\sigma$

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

- terminates in a finite number of iterations
- returns an $\varepsilon/2$-approximation of the optimal value function and an $\varepsilon$-optimal policy function (unless `iter_max` is reached)
While not explicit, in the actual implementation each algorithm is terminated if the number of iterations reaches \texttt{iter\_max}.

### Policy Iteration

The \texttt{DiscreteDP} policy iteration method runs as follows:

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

The policy iteration algorithm terminates in a finite number of iterations

It returns an optimal value function and an optimal policy function (unless \texttt{iter\_max} is reached).

### Modified Policy Iteration

The \texttt{DiscreteDP} modified policy iteration method runs as follows:

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

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

It returns an \(\varepsilon/2\)-approximation of the optimal value function and an \(\varepsilon\)-optimal policy function (unless \texttt{iter\_max} is reached).

See also the documentation for \texttt{DiscreteDP}.
These lectures look at important economic models that also illustrate common equilibrium concepts.

7.1 Schellings Segregation Model

7.1.1 Outline

In 1969, Thomas C. Schelling developed a simple but striking model of racial segregation [Sch69].

His model studies the dynamics of racially mixed neighborhoods.

Like much of Schelling's work, the model shows how local interactions can lead to surprising aggregate structure.

In particular, it shows that relatively mild preference for neighbors of similar race can lead in aggregate to the collapse of mixed neighborhoods, and high levels of segregation.

In recognition of this and other research, Schelling was awarded the 2005 Nobel Prize in Economic Sciences (joint with Robert Aumann).

In this lecture we (in fact you) will build and run a version of Schellings model.
7.1.2 The Model

We will cover a variation of Schellings model that is easy to program and captures the main idea.

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 \(0 < x, y < 1\).

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.

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.
7.1.3 Results

Let's have a look at the results we got when we coded and ran this model.

As discussed above, agents are initially mixed randomly together.

But after several cycles they become segregated into distinct regions.
7.1. Schellings Segregation Model

Cycle 2
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.
7.1.4 Exercises

Exercise 1

Implement and run this simulation for yourself.
Consider the following structure for your program.
Agents can be modeled as *objects*

Here is 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 is some pseudocode for the main loop:

```python
while agents are still moving
    for agent in agents
        give agent the opportunity to move
```

Use 250 agents of each type.

7.1.5 Solutions

Exercise 1

Here is 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.

```python
from random import uniform, seed
from math import sqrt
import matplotlib.pyplot as plt

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 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
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
            break
    if no_one_moved:
        break

print('Converged, terminating.')

Entering loop 1
Entering loop 2
7.1. Schellings Segregation Model

Entering loop 3
Entering loop 4
Converged, terminating.

7.2 A Lake Model of Employment and Unemployment

Contents

- A Lake Model of Employment and Unemployment
  - Overview
7.2.1 Overview

This lecture describes what has come to be called a lake model. The lake model is a basic tool for modeling unemployment. It allows us to analyze:

- flows between unemployment and employment
- how these flows influence steady state employment and unemployment rates

It is a good model for interpreting monthly labor department reports on gross and net jobs created and jobs destroyed.

The lakes in the model are the pools of employed and unemployed.

The flows between the lakes are caused by:

- firing and hiring
- entry and exit from the labor force

For the first part of this lecture, the parameters governing transitions into and out of unemployment and employment are exogenous.

Later, we will determine some of these transition rates endogenously using the McCall search model.

Well 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.

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.
7.2.2 The Model

The economy is inhabited by a very large number of ex ante identical workers.

The workers live forever, spending their lives moving between unemployment and employment.

Their rates of transition between employment and unemployment are governed by the following parameters:

- \( \lambda \), the job finding rate for currently unemployed workers
- \( \alpha \), 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 \( g = b - d \)

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)

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.

**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

**7.2.3 Implementation**

Let's code up these equations

To do this we're going to use a class that we call \( \text{LakeModel} \)

This class will

1. store the primitives \( \alpha, \lambda, b, d \)
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 an nice implementation, there's an issue we have to address

Derived data such as \( A \) depend on the primitives like \( \alpha \) and \( \lambda \)

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:

```python
import numpy as np

class LakeModel:
    '''
    Solves the lake model and computes dynamics of unemployment stocks and rates.
    
    Parameters:
    -----------
    \( \lambda \) : scalar
        The job finding rate for currently unemployed workers
    \( \alpha \) : 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, \( \lambda \)=0.283, \( \alpha \)=0.013, \( b \)=0.0124, \( d \)=0.00822):
        self._\( \lambda \), self._\( \alpha \), self._\( b \), self._\( d \) = \( \lambda \), \( \alpha \), \( b \), \( d \)
        self.compute_derived_values()

def compute_derived_values(self):
    # Unpack names to simplify expression
    \( \lambda \), \( \alpha \), \( b \), \( d \) = self._\( \lambda \), self._\( \alpha \), self._\( b \), self._\( d \)

    self._g = \( b \) - \( d \)
    self._A = np.array([[\((1-\( d \))*\((1-\lambda) + b\), \((1-\( d \))*\( \lambda \) + \(1-\( d \))*\( \alpha \) + \( b \)],
                         [\((1-\( d \))*\( \lambda \), \((1-\( d \))*\((1-\( \alpha \)))]
                        ))

    self._A_hat = self._A / (1 + self._g)

@property
def g(self):
    return self._g

@property
def A(self):
```
return self._A

@property
def A_hat(self):
    return self._A_hat

@property
def λ(self):
    return self._λ

@λ.setter
def λ(self, new_value):
    self._λ = new_value
    self.compute_derived_values()

@property
def a(self):
    return self._a

@a.setter
def a(self, new_value):
    self._a = 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`
    """
    x = 0.5 * np.ones(2)
    error = tol + 1
```python
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 the sequence of Employment and Unemployent 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 the sequence of employment and unemployent 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 $\alpha$, derived objects like $A$ will also change.
lm = LakeModel()
lm.α

0.013

lm.A

array([[ 0.72350626, 0.02529314],
       [ 0.28067374, 0.97888686]]

lm.α = 2
lm.A

array([[ 1.99596 , 0.72350626],
       [-0.99178 , 0.28067374]])

Aggregate Dynamics

Let's run a simulation under the default parameters (see above) starting from $X_0 = (12, 138)$

```python
import matplotlib.pyplot as plt

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(lm.simulate_stock_path(X_0, T))

axes[0].plot(X_path[:, 0], lw=2)
axes[0].set_title('Unemployment')

axes[1].plot(X_path[:, 1], lw=2)
axes[1].set_title('Employment')

axes[2].plot(X_path.sum(1), lw=2)
axes[2].set_title('Labor force')

for ax in axes:
    ax.grid()

plt.tight_layout()
plt.show()
```
The aggregates $E_t$ and $U_t$ don't converge because their sum $E_t + U_t$ grows at rate $g$.

On the other hand, the vector of employment and unemployment rates $x_t$ can be in a steady state $\bar{x}$ if there exists an $\bar{x}$ such that

- $\bar{x} = \hat{A}\bar{x}$
- the components satisfy $\bar{e} + \bar{u} = 1$

This equation tells us that a steady state level $\bar{x}$ is an eigenvector of $\hat{A}$ associated with a unit eigenvalue. We also have $x_t \to \bar{x}$ as $t \to \infty$ provided that the remaining eigenvalue of $\hat{A}$ has modulus less than 1.

This is the case for our default parameters:

```python
lm = LakeModel()
e, f = np.linalg.eigvals(lm.A_hat)
abs(e), abs(f)
```

(0.69530673783584618, 1.0)

7.2. A Lake Model of Employment and Unemployment
Let's look at the convergence of the unemployment and employment rate to steady state levels (dashed red line)

```python
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(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()
```
7.2.4 Dynamics of an Individual Worker

An individual workers 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 $\psi_t$ 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 $\psi_t$ 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 $\psi^*$

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.

**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.

**Convergence rate**

How long does it take for time series sample averages to converge to cross sectional averages?

We can use QuantEcon.pys MarkovChain class to investigate this.

Let's plot the path of the sample averages over 5,000 periods.
from quantecon import MarkovChain

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

### 7.2.5 Endogenous Job Finding Rate

We now make the hiring rate endogenous
The transition rate from unemployment to employment will be determined by the McCall search model

\[ \text{[McC70]} \]

All details relevant to the following discussion can be found in *our treatment* of that model

**Reservation Wage**

The most important thing to remember about the model is that optimal decisions are characterized by a reservation wage $\bar{w}$

---

804 Chapter 7. Multiple Agent Models
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:

- \( \alpha \), the separation rate
- \( \beta \), the discount factor
- \( \gamma \), the offer arrival rate
- \( c \), unemployment compensation

**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 \( \alpha \).

But their optimal decision rules determine the probability \( \lambda \) of leaving unemployment.

This is now

\[
\lambda = \gamma \mathbb{P}\{w_t \geq \bar{w}\} = \gamma \sum_{w' \geq \bar{w}} p(w') \tag{7.1}
\]

**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 \( \tau \) 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 \((c, \tau)\) of government policy, we can solve for the workers optimal reservation wage.

This determines \( \lambda \) via (7.1) 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

$$W := e \mathbb{E}[V \mid \text{employed}] + u U$$

where the notation $V$ and $U$ is as defined in the McCall search model lecture.

The wage offer distribution will be a discretized version of the lognormal distribution $LN(\log(20), 1)$, as shown in the next figure.

We take a period to be a month.

We set $b$ and $d$ to match monthly birth and death rates, respectively, in the U.S. population.

- $b = 0.0124$
- $d = 0.00822$

Following [DFH06], we set $\alpha$, the hazard rate of leaving employment, to

- $\alpha = 0.013$

**Fiscal Policy Code**

We will make use of code we wrote in the McCall model lecture, embedded below for convenience.
The first piece of code, repeated below, implements value function iteration

```python
import numpy as np
from quantecon.distributions import BetaBinomial
from numba import jit

# 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.σ = 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
        self.w_vec = np.linspace(10, 20, n)  # wages between 10 and 20
        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):
```

7.2. A Lake Model of Employment and Unemployment 807
# 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 + \( β * γ * \text{np.sum(np.maximum(U, V) * p_vec)} \)

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))
    V_new = np.empty_like(V)
    U = 1
    i = 0
    error = tol + 1
    while error > tol and i < max_iter:
        U_new = _update_bellman(mcm.a, mcm.b, mcm.g, mcm.c, mcm.sigma, 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

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.
    """

The second piece of code repeated from *the McCall model lecture* is used to complete the reservation wage.
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

```python
from scipy.stats import norm
from scipy.optimize import brentq

# 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(0, 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, \tau):
    ""
    Compute the reservation wage, job finding rate and value functions of the
    workers given c and \tau.
    """

    mcm = McCallModel(\alpha=\alpha_q,
    \beta=\beta,
    \gamma=\gamma,
    c=c-\tau,  # post tax compensation
    \sigma=\sigma,
    w_vec=w_vec-\tau,  # post tax wages
    p_vec=p_vec)

    w_bar, V, U = compute_reservation_wage(mcm, return_values=True)
    \lambda = \gamma * np.sum(p_vec[w_vec - \tau > w_bar])
    return w_bar, \lambda, V, U

def compute_steady_state_quantities(c, \tau):
    ""
    Compute the steady state unemployment rate given c and \tau using optimal
    quantities from the McCall model and computing corresponding steady state
    quantities
    """

    w_bar, \lambda, V, U = compute_optimal_quantities(c, \tau)

    # Compute steady state employment and unemployment rates
    lm = LakeModel(\alpha=\alpha_q, \lambda=\lambda, b=b, d=d)
    x = lm.rate_steady_state()
    u, e = x

    # Compute steady state welfare
    w = np.sum(V * p_vec * (w_vec - \tau > w_bar)) / np.sum(p_vec * (w_vec -
    \tau > w_bar))
    welfare = e * w + u * U

    return e, u, welfare

def find_balanced_budget_tax(c):
    ""
    Find 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

    \tau = brentq(steady_state_budget, 0.0, 0.9 * c)
    return \tau
# 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()

The figure that the preceding code listing generates is shown below
Welfare first increases and then decreases as unemployment benefits rise.

The level that maximizes steady state welfare is approximately 62.

### 7.2.6 Exercises

**Exercise 1**

Consider an economy with 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 $\alpha$ and $\lambda$ follow [DFH06])

Suppose that in response to new legislation the hiring rate reduces to $\lambda = 0.2$

Plot the transition dynamics of the unemployment and employment stocks for 50 periods

Plot the transition dynamics for the rates

How long does the economy take to converge to its new steady state?

What is the new steady state level of employment?

**Exercise 2**

Consider an economy with 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?

### 7.2.7 Solutions

### 7.2.8 Lake Model Solutions

**Exercise 1**

We begin by constructing the class containing the default parameters and assigning the steady state values to $x_0$

```python
lm = LakeModel()
x0 = lm.rate_steady_state()
print(f"Initial Steady State: {x0}")
```

Initial Steady State: [ 0.08266806 0.91733194]

Initialize the simulation values

```python
N0 = 100
T = 50
```

New legislation changes $\lambda$ to 0.2

```python
lm.lmda = 0.2

xbar = lm.rate_steady_state()  # new steady state
X_path = np.vstack(lm.simulate_stock_path(x0 * N0, T))
```
\[ x_{\text{path}} = \text{np.vstack(} \text{lm.simulate_rate_path}(x0, T) \text{)} \]
\[ \text{print(} \text{f"New Steady State: \{xbar\}"} \) \]

```
New Steady State: [ 0.11309573  0.88690427]
```

Now plot stocks

```
fig, axes = plt.subplots(3, 1, figsize=[10, 9])

axes[0].plot(X_path[:, 0])
axes[0].set_title('Unemployment')

axes[1].plot(X_path[:, 1])
axes[1].set_title('Employment')

axes[2].plot(X_path.sum(1))
axes[2].set_title('Labor force')

for ax in axes:
    ax.grid()

plt.tight_layout()
plt.show()
```
And how the rates evolve

```python
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()
```

7.2. A Lake Model of Employment and Unemployment
We see that it takes 20 periods for the economy to converge to its new steady state levels

**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.

```python
lm = LakeModel()
x0 = lm.rate_steady_state()
```

Here are the other parameters:

```python
b_hat = 0.003
T_hat = 20
```

Let's increase $b$ to the new value and simulate for 20 periods.
\texttt{lm.b = b\_hat}
\texttt{X\_path1 = np.vstack(lm.simulate_stock_path(x0 * N0, T\_hat)) \# simulate stocks}
\texttt{x\_path1 = np.vstack(lm.simulate_rate_path(x0, T\_hat)) \# simulate rates}

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

\texttt{lm.b = 0.0124}
\texttt{X\_path2 = np.vstack(lm.simulate_stock_path(X\_path1[-1, :2], T-T\_hat+1)) \# simulate stocks}
\texttt{x\_path2 = np.vstack(lm.simulate_rate_path(x\_path1[-1, :2], T-T\_hat+1)) \# simulate rates}

Finally we combine these two paths and plot

\texttt{x\_path = np.vstack([x\_path1, x\_path2[1:]])) \# note [1:] to avoid doubling period 20}
\texttt{X\_path = np.vstack([X\_path1, X\_path2[1:]]))}

\texttt{fig, axes = plt.subplots(3, 1, figsize=[10, 9])}
\texttt{axes[0].plot(X\_path[:, 0])}
\texttt{axes[0].set_title('Unemployment')}
\texttt{axes[1].plot(X\_path[:, 1])}
\texttt{axes[1].set_title('Employment')}
\texttt{axes[2].plot(X\_path.sum(1))}
\texttt{axes[2].set_title('Labor force')}

\texttt{for ax in axes:}
\texttt{\hspace{1em}ax.grid()}

\texttt{plt.tight_layout()}
\texttt{plt.show()}

\section*{7.2. A Lake Model of Employment and Unemployment}
And the rates

```python
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()
```
7.3 Rational Expectations Equilibrium

Contents

- Rational Expectations Equilibrium
  - Overview
  - Defining Rational Expectations Equilibrium
  - Computation of an Equilibrium
  - Exercises
  - Solutions

If you’re so smart, why aren’t you rich?

7.3.1 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 [LP71]
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 agents problem differs from a planners, 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 $y$

**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 individuals 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 sells output $y$
The price \( p \) of the good lies on an inverse demand curve

\[
p = a_0 - a_1 Y
\]  

(7.2)

where

- \( a_i > 0 \) for \( i = 0, 1 \)
- \( Y = ny \) is the market-wide level of output

Each firm has total cost function

\[
c(y) = c_1 y + 0.5 c_2 y^2, \quad c_i > 0 \text{ for } i = 1, 2
\]

The profits of a representative firm are \( py - c(y) \)

Using (7.2), we can express the problem of the representative firm as

\[
\max_y \left[ (a_0 - a_1 Y) y - c_1 y - 0.5 c_2 y^2 \right]
\]  

(7.3)

In posing problem (7.3), 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 (7.3)

This assures that the firm is a price taker

The first order condition for problem (7.3) is

\[
a_0 - a_1 Y - c_1 - c_2 y = 0
\]  

(7.4)

At this point, but not before, we substitute \( Y = ny \) into (7.4) to obtain the following linear equation

\[
a_0 - c_1 - (a_1 + n^{-1} c_2) Y = 0
\]  

(7.5)

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 (7.2)

**Further Reading**

References for this lecture include

- [LP71]
- [Sar87], chapter XIV
- [LS18], chapter 7
7.3.2 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.

**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 sells output \(y_t\).

The price \(p_t\) of the good lies on the inverse demand curve

\[
p_t = a_0 - a_1 Y_t
\]

where

- \(a_i > 0\) for \(i = 0, 1\)
- \(Y_t = ny_t\) is the market-wide level of output

**The Firms 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

\[
sum_{t=0}^{\infty} \beta^t r_t
\]

where
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 firms 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 (7.6), the firms 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 Firms Beliefs**

We suppose the firm believes that market-wide output $Y_t$ follows the law of motion

$$Y_{t+1} = H(Y_t) \tag{7.9}$$

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 lets fix a particular belief $H$ in (7.9) and investigate the firms response to it

Let $v$ be the optimal value function for the firms problem given $H$

The value function satisfies the Bellman equation

$$r_t := p_t y_t - \frac{\gamma (y_{t+1} - y_t)^2}{2}, \quad y_0 \text{ given} \tag{7.8}$$
\[ v(y, Y) = \max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma (y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \]  

(7.10)

Let's denote the firms optimal policy function by \( h \), so that

\[ y_{t+1} = h(y_t, Y_t) \]  

(7.11)

where

\[ h(y, Y) := \arg \max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma (y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \]  

(7.12)

Evidently \( v \) and \( h \) both depend on \( H \)

**First-Order Characterization of \( h \)**

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

\[ -\gamma (y' - y) + \beta v_y(y', H(Y)) = 0 \]  

(7.13)

An important useful envelope result of Benveniste-Scheinkman \([BS79]\) implies that to differentiate \( v \) with respect to \( y \) we can naively differentiate the right side of (7.10), giving

\[ v_y(y, Y) = a_0 - a_1 Y + \gamma (y' - y) \]

Substituting this equation into (7.13) gives the **Euler equation**

\[ -\gamma (y_{t+1} - y_t) + \beta [a_0 - a_1 Y_{t+1} + \gamma (y_{t+2} - y_{t+1})] = 0 \]  

(7.14)

The firm optimally sets an output path that satisfies (7.14), taking (7.9) as given, and subject to

- the initial conditions for \((y_0, Y_0)\)
- the terminal condition \( \lim_{t \to \infty} \beta^t v_y(y_t, Y_t) = 0 \)

This last condition is called the **transversality condition**, and acts as a first-order necessary condition at infinity.

The firms decision rule solves the difference equation (7.14) subject to the given initial condition \( y_0 \) and the transversality condition.

Note that solving the Bellman equation (7.10) for \( v \) and then \( h \) in (7.12) yields a decision rule that automatically imposes both the Euler equation (7.14) and the transversality condition.
The Actual Law of Motion for \( \{Y_t\} \)

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

\[
Y_{t+1} = nh(Y_t/n, Y_t)
\]  
(7.15)

Thus, when firms believe that the law of motion for market-wide output is \((7.9)\), their optimizing behavior makes the actual law of motion be \((7.15)\)

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 firms 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 \((7.9)\) and \((7.15)\)

Fixed point characterization

As we've seen, the firms optimum problem induces a mapping \( \Phi \) from a perceived law of motion \( H \) for market-wide output to an actual law of motion \( \Phi(H) \)
The mapping \( \Phi \) is the composition of two operations, taking a perceived law of motion into a decision rule via \((7.10)\)–\((7.12)\), and a decision rule into an actual law via \((7.15)\)
The \( H \) component of a rational expectations equilibrium is a fixed point of \( \Phi \)

7.3.3 Computation of an Equilibrium

Now let's consider the problem of computing the rational expectations equilibrium

Misbehavior of \( \Phi \)

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 \( \Phi \)
Unfortunately, the mapping \( \Phi \) is not a contraction
In particular, there is no guarantee that direct iterations on \( \Phi \) converge\(^1\)

\(^1\) 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 \( \Phi \) that can be approximated as \( \gamma \Phi + (1 - \gamma)I \). Here \( I \) is the identity operator and \( \gamma \in (0, 1) \) is a relaxation parameter. See [MS89] and [EH01] for statements and applications of this approach to establish conditions under which collections of adaptive agents who use least squares learning converge to a rational expectations equilibrium.
Fortunately, there is another method that works here. The method exploits a general connection between equilibrium and Pareto optimality expressed in the fundamental theorems of welfare economics (see, e.g., [MCWG95]). Lucas and Prescott [LP71] used this method to construct a rational expectations equilibrium. The details follow.

**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 well 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 \( t \)

\[
s(Y_t, Y_{t+1}) := \int_0^{Y_t} (a_0 - a_1 x) \, dx - \frac{\gamma(Y_{t+1} - Y_t)^2}{2}
\]

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 \( Y_0 \).

**Solution of the Planning Problem**

Evaluating the integral in (7.16) 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

\[
V(Y) = \max_Y \left\{ a_0 Y - \frac{a_1}{2} Y^2 - \frac{\gamma(Y' - Y)^2}{2} + \beta V(Y') \right\}
\]

The associated first order condition is

\[
-\gamma(Y' - Y) + \beta V'(Y') = 0
\]
Applying the same Benveniste-Scheinkman formula gives

\[ V'(Y) = a_0 - a_1 Y + \gamma(Y' - Y) \]

Substituting this into equation (7.18) and rearranging leads to the Euler equation

\[ \beta a_0 + \gamma Y_t - [\beta a_1 + \gamma(1 + \beta)] Y_{t+1} + \gamma \beta Y_{t+2} = 0 \]  

(7.19)

**The Key Insight**

Return to equation (7.14) 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 (7.19) and (7.14) 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 \( Y_{t+1} = n_0 y_t \) 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 (7.17)

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 planners problem is an LQ problem implies an optimal policy and hence aggregate law of motion taking the form

\[ Y_{t+1} = \kappa_0 + \kappa_1 Y_t \]  

(7.20)

for some parameter pair \( \kappa_0, \kappa_1 \)

Now that we know the aggregate law of motion is linear, we can see from the firms Bellman equation (7.10) that the firms problem can also be framed as an LQ problem

As you're asked to show in the exercises, the LQ formulation of the firms problem implies a law of motion that looks as follows
\[ y_{t+1} = h_0 + h_1 y_t + h_2 Y_t \]  

(7.21)

Hence a rational expectations equilibrium will be defined by the parameters \((\kappa_0, \kappa_1, h_0, h_1, h_2)\) in (7.20)–(7.21)

### 7.3.4 Exercises

**Exercise 1**

Consider the firm problem *described above*

Let the firms belief function \(H\) be as given in (7.20)

Formulate the firms 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 firms 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 firms problem in the form (7.21) and give the values for each \(h_j\)

If there were \(n\) identical competitive firms all behaving according to (7.21), what would (7.21) imply for the actual law of motion (7.9) for market supply

**Exercise 2**

Consider the following \(\kappa_0, \kappa_1\) pairs as candidates for the aggregate law of motion component of a rational expectations equilibrium (see (7.20))

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)

**Exercise 3**

Recall the planners problem *described above*

1. Formulate the planners problem as an LQ problem
2. Solve it using the same parameter values in exercise 1

\[ a_0 = 100, \quad a_1 = 0.05, \quad \beta = 0.95, \quad \gamma = 10 \]

3. Represent the solution in the form \( Y_{t+1} = \kappa_0 + \kappa_1 Y_t \)

4. Compare your answer with the results from exercise 2

**Exercise 4**

A monopolist faces the industry demand curve (7.6) 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.

### 7.3.5 Solutions

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

Well use the LQ class from quantecon

```from quantecon import LQ```

**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

\[ x_t = \begin{bmatrix} y_t \\ Y_t \\ 1 \end{bmatrix}, \quad u_t = y_{t+1} - y_t \]

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
• $x_t'Rx_t + u_t'Qu_t = -r_t$
• $x_{t+1} = Ax_t + Bu_t$

Well use the module lqcontrol.py to solve the firms problem at the stated parameter values

This will return an LQ policy $F$ with the interpretation $u_t = -Fx_t$, or

$$y_{t+1} - y_t = -F_0y_t - F_1Y_t - F_2$$

Matching parameters with $y_{t+1} = h_0 + h_1y_t + h_2Y_t$ leads to

$$h_0 = -F_2, \quad h_1 = 1 - F_0, \quad h_2 = -F_1$$

Here is our solution

```python
# == 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], [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

$$y_{t+1} = 96.949 + y_t - 0.046Y_t$$
For the case \( n > 1 \), recall that \( Y_t = ny_t \), which, combined with the previous equation, yields
\[
Y_{t+1} = n (96.949 + y_t - 0.046 Y_t) = n96.949 + (1 - n0.046)Y_t
\]

**Exercise 2**

To determine whether a \( \kappa_0, \kappa_1 \) 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

```python
candidates = ((94.0886298678, 0.923409232937),
              (93.2119845412, 0.984323478873),
              (95.0818452486, 0.952459076301))

for \( \kappa_0, \kappa_1 \) in candidates:
    # == Form the associated law of motion == #
    A = np.array([[1, 0, 0], [0, \kappa_1, \kappa_0], [0, 0, 1]])

    # == Solve the LQ problem for the firm == #
    lq = LQ(Q, R, A, B, beta=\beta)
    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((\kappa_0, \kappa_1), (h0, h1 + h2)):
        print(f'Equilibrium pair = (\kappa_0), (\kappa_1)\')
        print(f'h0, h1, h2 = 95.0818452486 0.952459076301')
        break

Equilibrium pair = 95.0818452486 0.952459076301
(h0, h1, h2) = 95.0818910013 1.0 -0.047540944278
```

The output tells us that the answer is pair (iii), which implies \( (h_0, h_1, h_2) = (95.0819, 1.0000, -0.0475) \)

(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 \( (\kappa_0, \kappa_1) \) pair to the associated firm law and then to a new \( (\kappa_0, \kappa_1) \) pair

This amounts to implementing the operator \( \Phi \) described in the lecture

**7.3. Rational Expectations Equilibrium** 831
(There is in general no guarantee that this iterative process will converge to a rational expectations equilibrium)

**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

- \( x_t'Rx_t + u_t'Qu_t = -s(Y_t, Y_{t+1}) \)
- \( x_{t+1} = Ax_t + Bu_t \)

By obtaining the optimal policy and using \( u_t = -Fx_t \) or

\[
Y_{t+1} - Y_t = -F_0Y_t - F_1
\]

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:

```python
# == 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 = \gamma / 2

# == Solve for the optimal policy == #
lq = LQ(Q, R, A, B, beta=\beta)
P, F, d = lq.stationary_values()

# == Print the results == #
F = F.flatten()
k0, k1 = -F[1], 1 - F[0]
print(k0, k1)
```

95.0818745921 0.952459062704

The output yields the same \((\kappa_0, \kappa_1)\) pair obtained as an equilibrium from the previous exercise.

---

Chapter 7. Multiple Agent Models
Exercise 4

The monopolists LQ problem is almost identical to the planners 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

```python
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)
F, d = lq.stationary_values()
F = F.flatten()
m0, m1 = -F[1], 1 - F[0]
p
```

We see that the law of motion for the monopolist is approximately \( Y_{t+1} = 73.4729 + 0.9265Y_t \)

In the rational expectations case the law of motion was approximately \( Y_{t+1} = 95.0818 + 0.9525Y_t \)

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 \( Y_{t+1} = c_0 + c_1Y_t \), 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

7.4 Markov Perfect Equilibrium

7.4.1 Overview

This lecture describes the concept of Markov perfect equilibrium

Markov perfect equilibrium is a key notion for analyzing economic problems involving dynamic strategic interaction, and a cornerstone of applied game theory

In this lecture we teach Markov perfect equilibrium by example

We will focus on settings with

- two players
• quadratic payoff functions
• linear transition rules for the state

Other references include chapter 7 of [LS18]

7.4.2 Background

Markov perfect equilibrium is a refinement of the concept of Nash equilibrium

It is used to study settings where multiple decision makers interact non-cooperatively over time, each seeking to pursue its own objective

The agents in the model face a common state vector, the time path of which is influenced by – and influences – their decisions

In particular, the transition law for the state that confronts each agent is affected by decision rules of other agents

Individual payoff maximization requires that each agent solve a dynamic programming problem that includes this transition law

Markov perfect equilibrium prevails when no agent wishes to revise its policy, taking as given the policies of all other agents

Well known examples include
• Choice of price, output, location or capacity for firms in an industry (e.g., [EP95], [Rya12], [DS10])
• Rate of extraction from a shared natural resource, such as a fishery (e.g., [LM80], [VL11])

Let’s examine a model of the first type

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

\[ p = a_0 - a_1 (q_1 + q_2) \]  (7.22)

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 (7.22) 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:
\[ \pi_i = pq_i - \gamma (\hat{q}_i - q_i)^2, \quad \gamma > 0, \quad (7.23) \]

Substituting the inverse demand curve \((7.22)\) into \((7.23)\) lets us express the one-period payoff as

\[ \pi_i(q_i, q_{-i}, \hat{q}_i) = a_0 q_i - a_1 q_i^2 - a_1 q_i q_{-i} - \gamma (\hat{q}_i - q_i)^2, \quad (7.24) \]

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

\[ v_i(q_i, q_{-i}) = \max_{\hat{q}_i} \{ \pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i(\hat{q}_i, f_{-i}(q_{-i}, q_i)) \} \quad (7.25) \]

**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 the Bellman equation \((7.25)\)
- The maximizer on the right side of \((7.25)\) is equal to \(f_i(q_i, q_{-i})\)

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.

**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

\[ v_i^{j+1}(q_i, q_{-i}) = \max_{\hat{q}_i} \{ \pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i^j(\hat{q}_i, f_{-i}(q_{-i}, q_i)) \} \quad (7.26) \]

These iterations can be challenging to implement computationally.

However, they simplify for the case in which the one-period payoff functions are quadratic and the transition laws are linear which takes us to our next topic.
7.4.3 Linear Markov perfect equilibria

As we saw in the duopoly example, the study of Markov perfect equilibria in games with two players leads us to an interrelated pair of Bellman equations.

In linear quadratic dynamic games, these stacked Bellman equations become stacked Riccati equations with a tractable mathematical structure.

We lay out that structure in a general setup and then apply it to some simple problems.

**Coupled linear regulator problems**

We consider a general linear quadratic regulator game with two players.

For convenience, we will 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

\[
\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{ x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2 x_t' W_i u_{it} + 2 u_{-it}' M_i u_{it} \right\}
\]

(7.27)

while the state evolves according to

\[
x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t}
\]

(7.28)

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 $n \times k_i$

**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 (7.27) and (7.28), then player 1’s problem becomes minimization of

\[
\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{ x_t' \Pi_{1t} x_t + u_{1t}' Q_1 u_{1t} + 2u_{1t}' \Gamma_{1t} x_t \right\}
\]

(7.29)

subject to

\[
x_{t+1} = \Lambda_{1t} x_t + B_{1t} u_{1t},
\]

(7.30)

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.

The policy rule that solves this problem is

\[
F_{1t} = (Q_1 + \beta B_{1t}' P_{1t+1} B_1)^{-1}(\beta B_{1t}' P_{1t+1} \Lambda_{1t} + \Gamma_{1t})
\]

(7.31)

where \( P_{1t} \) solves the matrix Riccati difference equation

\[
P_{1t} = \Pi_{1t} - (\beta B_{1t}' P_{1t+1} \Lambda_{1t} + \Gamma_{1t})'(Q_1 + \beta B_{1t}' P_{1t+1} B_1)^{-1}(\beta B_{1t}' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) + \beta \Lambda_{1t}' P_{1t+1} \Lambda_{1t}
\]

(7.32)

Similarly, the policy that solves player 2’s problem is

\[
F_{2t} = (Q_2 + \beta B_{2t}' P_{2t+1} B_2)^{-1}(\beta B_{2t}' P_{2t+1} \Lambda_{2t} + \Gamma_{2t})
\]

(7.33)

where \( P_{2t} \) solves

\[
P_{2t} = \Pi_{2t} - (\beta B_{2t}' P_{2t+1} \Lambda_{2t} + \Gamma_{2t})'(Q_2 + \beta B_{2t}' P_{2t+1} B_2)^{-1}(\beta B_{2t}' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) + \beta \Lambda_{2t}' P_{2t+1} \Lambda_{2t}
\]

(7.34)

Here in all cases \( t = t_0, \ldots, t_1 - 1 \) and the terminal conditions are \( P_{1t_1} = 0 \).

The solution procedure is to use equations (7.31), (7.32), (7.33), and (7.34), and work backwards from time \( t_1 - 1 \).

Since we were working backwards, \( P_{1t+1} \) and \( P_{2t+1} \) are taken as given at each stage.

Moreover, since

7.4. Markov Perfect Equilibrium
• some terms on the right hand side of (7.31) contain $F_{2t}$
• some terms on the right hand side of (7.33) contain $F_{1t}$
we need to solve these $k_1 + k_2$ equations simultaneously

**Key insight**

A key insight is that equations (7.31) and (7.33) 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 (7.32) and (7.34)

**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

**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

**7.4.4 Application**

Lets use these procedures to treat some applications, starting with the duopoly model

**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 (7.24).

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:

$$
x_{t+1} = (A - B_1 F_1 - B_2 F_2) x_t \quad (7.35)
$$

### 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:

```python
""
@authors: Chase Coleman, Thomas Sargent, John Stachurski
""
import numpy as np
import quantecon as qe

# == Parameters == #
a0 = 10.0
a1 = 2.0
beta = 0.96
gamma = 12.0

# == In LQ form == #
A = np.eye(3)
B1 = np.array([[0.], [1.], [0.]])
B2 = np.array([[0.], [0.], [1.]]).

R1 = np.array([[0., -a0 / 2., 0.],
               [-a0 / 2., a1, a1 / 2.],
               [0, a1 / 2., 0]])
R2 = np.array([[0., 0., -a0 / 2],
               [0., 0., 0.],
               [0., 0., 0.]])
```

7.4. Markov Perfect Equilibrium
# == 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:

")
print(f"F1 = 
{F1}"")
print(f"F2 = 
{F2}"")
print("\n")

Running the code produces the following output

Computed policies for firm 1 and firm 2:

F1 = 
[-0.66846615 0.29512482 0.07584666]
F2 = 
[-0.66846615 0.07584666 0.29512482]

One way to see that $F_i$ is indeed optimal for firm $i$ taking $F_j$ as given is to use QuantEcon.pys $LQ$ class

In particular, lets take $F_2$ as computed above, plug it into (7.29) and (7.30) to get firm 1s problem and solve it using $LQ$

We hope that the resulting policy will agree with $F_1$ as computed above

```
A1 = A - B2 @ F2
lq1 = qe.LQ(Q1, R1, A1, B1, beta=β)
F1_ih, F1_ih, d = lq1.stationary_values()
F1_ih
```

array([[-0.66846611, 0.29512481, 0.07584666]])

This is close enough for rock and roll, as they say in the trade

Indeed, `np.allclose` agrees with our assessment

```python
np.allclose(F1, F1_ih)
```

```
True
```

**Dynamics**

Lets now investigate the dynamics of price and output in this simple duopoly model under the MPE policies

Given our optimal policies $F_1$ and $F_2$, the state evolves according to (7.35)
The following program

- imports $F_1$ and $F_2$ from the previous program along with all parameters
- computes the evolution of $x_t$ using (7.35)
- extracts and plots industry output $q_t = q_{1t} + q_{2t}$ and price $p_t = a_0 - a_1 q_t$

```python
import matplotlib.pyplot as plt

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 $q_{10} = q_{20} = 1.0$

The resulting figure looks as follows
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.
7.4.5 Exercises

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 LQ class

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 [Jud90]

Two firms set prices and quantities of two goods interrelated through their demand curves

Relevant variables are defined as follows:

- $I_{it}$ = inventories of firm $i$ at beginning of $t$
- $q_{it}$ = production of firm $i$ during period $t$
- $p_{it}$ = price charged by firm $i$ during period $t$
- $S_{it}$ = sales made by firm $i$ during period $t$
- $E_{it}$ = costs of production of firm $i$ during period $t$
- $C_{it}$ = costs of carrying inventories for firm $i$ during $t$

The firms cost functions are

- $C_{it} = c_{i1} + c_{i2}I_{it} + 0.5c_{i3}I_{it}^2$
- $E_{it} = e_{i1} + e_{i2}q_{it} + 0.5e_{i3}q_{it}^2$ 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

$$S_t = Dp_{it} + b$$

where

- $S_t = [S_{1t} \quad S_{2t}]'$
- $D$ is a $2 \times 2$ 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 Judds 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

\[
\begin{align*}
\delta &= 0.02 \\
D &= \text{np.array}([[\text{-1}, 0.5], [0.5, \text{-1}]]) \\
b &= \text{np.array}([25, 25]) \\
c1 = c2 &= \text{np.array([1, \text{-2}, 1])} \\
e1 = e2 &= \text{np.array([10, 10, 3])}
\end{align*}
\]

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.

7.4. Markov Perfect Equilibrium
7.4.6 Solutions

Exercise 1

First let's compute the duopoly MPE under the stated parameters

```python
# == 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.],
               [-a0 / 2., a1, a1 / 2.],
               [ 0, a1 / 2., 0.]])
B2 = np.array([[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 == #
P1, P2, F1, F2 = qe.nnash(A, B1, B2, R1, R2, Q1,
                          Q2, S1, S2, W1, W2, M1, M2, beta=β)
```
Now we evaluate the time path of industry output and prices given initial condition \( q_{10} = q_{20} = 1 \)

\[
AF = A - B_1 @ F_1 - B_2 @ F_2
\]
\[
n = 20
\]
\[
x = np.empty((3, n))
x[:, 0] = 1, 1, 1
\]
\[
\textbf{for } t \textbf{ in range} \textbf{(n-1)}: 
    x[:, t+1] = AF @ x[:, t]
\]
\[
q1 = x[1, :]
q2 = x[2, :]
q = q1 + q2  \quad \# \text{Total output, MPE}
\]
\[
p = a0 - a1 * q \quad \# \text{Price, MPE}
\]

Next let's have a look at the monopoly solution

For the state and control we take

\[
x_t = q_t - \bar{q} \quad \text{and} \quad u_t = q_{t+1} - q_t
\]

To convert to an LQ problem we set

\[
R = a_1 \quad \text{and} \quad Q = \gamma
\]

in the payoff function \( x'_t R x_t + u'_t Q u_t \) and

\[
A = B = 1
\]

in the law of motion \( x_{t+1} = Ax_t + Bu_t \)

We solve for the optimal policy \( u_t = -Fx_t \) and track the resulting dynamics of \( \{q_t\} \), starting at \( q_0 = 2.0 \)

\[
R = a1
\]
\[
Q = \gamma
\]
\[
A = B = 1
\]
\[
lq_alt = qe.LQ(Q, R, A, B, beta=\beta)
\]
\[
F, F, d = lq_alt.stationary_values()
\]
\[
q_bar = a0 / (2.0 * a1)
\]
\[
qm = np.empty(n)
\]
\[
qm[0] = 2
\]
\[
x0 = qm[0] - q_bar
\]
\[
x = x0
\]
\[
\textbf{for } i \textbf{ in range}(1, n):
    x = A * x - B * F * x
    qm[i] = float(x) + q_bar
\]
\[
pm = a0 - a1 * qm
\]

Let's have a look at the different time paths

\[
fig, axes = plt.subplots(2, 1, figsize=(9, 9))
\]
\[
ax = axes[0]
ax.plot(qm, 'b-', lw=2, alpha=0.75, label='monopolist output')
ax.plot(q, 'g-', lw=2, alpha=0.75, label='MPE total output')
Exercise 2

We treat the case $\delta = 0.02$
Recalling that the control and state are

\[
u_t = \begin{bmatrix} p_{it} \\ q_{it} \end{bmatrix} \quad \text{and} \quad x_t = \begin{bmatrix} I_{1t} \\ I_{2t} \\ 1 \end{bmatrix}
\]

we set up the matrices as follows:

```python
# == Create matrices needed to compute the Nash feedback equilibrium == #
A = np.array([[\delta_1, 0, -\delta_1 * b[0]],
               [0, \delta_1, -\delta_1 * b[1]],
               [0, 0, 1]])

B1 = \delta_1 * np.array([[1, -D[0, 0]],
                          [0, -D[1, 0]],
                          [0, 0]])

B2 = \delta_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, 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, 0.5 * e1[1], b[0] / 2.1]])

W2 = np.array([[0, 0],
               [0, 0],
               [0.5 * e2[1], b[1] / 2.1]])

M1 = np.array([[0, 0], [0, D[0, 1] / 2.]])

M2 = np.copy(M1)
```

We can now compute the equilibrium using `qe.nnash`
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2)

print("Firm 1's feedback rule:")
print(F1)

print("Firm 2's feedback rule:")
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 lets look at the dynamics of inventories, and reproduce the graph corresponding to $\delta = 0.02$

AF = A - B1 @ F1 - B2 @ F2
n = 25
x = np.empty((3, n))
x[:, 0] = 2, 0, 1
for t in range(n-1):
    x[:, t+1] = AF @ x[:, t]
I1 = x[0, :]
I2 = x[1, :]
fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(I1, 'b-', lw=2, alpha=0.75, label='inventories, firm 1')
ax.plot(I2, 'g-', lw=2, alpha=0.75, label='inventories, firm 2')
ax.set_title(rf'\$\delta = {\delta}\$')
ax.legend()
plt.show()
7.5 Robust Markov Perfect Equilibrium

Co-author: Dongchen Zou

7.5.1 Overview

This lecture describes a Markov perfect equilibrium with robust agents. We focus on special settings with

- two players
- quadratic payoff functions
- linear transition rules for the state vector

These specifications simplify calculations and allow us to give a simple example that illustrates basic forces. This lecture is based on ideas described in chapter 15 of [HS08] and in *Markov perfect equilibrium* and *Robustness*.

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 backwards 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 will construct robust firms version of the classic duopoly model with adjustment costs analyzed in *Markov perfect equilibrium*

### 7.5.2 Linear Markov perfect equilibria with robust agents

As we saw in *Markov perfect equilibrium*, the study of Markov perfect equilibria in dynamic games with two players leads us to an interrelated pair of Bellman equations

In linear quadratic dynamic games, these stacked Bellman equations become stacked Riccati equations with a tractable mathematical structure

**Modified coupled linear regulator problems**

We consider a general linear quadratic regulator game with two players, each of whom fears model misspecification.

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 $C v_{it}$ 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, well 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

\[
\sum_{t=t_0}^{t_1} \beta^{t-t_0} \left\{ x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{it}' S_i u_{it} + 2 x_t' W_i u_{it} + 2 u_{it}' M_i u_{it} - \theta_i v_{it}' v_{it} \right\}
\]

(7.36)

while thinking that the state evolves according to

\[
x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t} + C v_{it}
\]

(7.37)

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 \( n \times h \)
- \( \theta_i \in [\bar{\theta}_i, +\infty) \) 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

**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 (7.36) and (7.37), then player 1's problem becomes minimization-maximization of

\[
\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{ x_t' \Pi_{1t} x_t + u_{1t}'Q_1 u_{1t} + 2u_{1t}' \Gamma_{1t} x_t - \theta_1 v_{1t}^1 v_{1t} \right\}
\]

subject to

\[
x_{t+1} = \Lambda_{1t} x_t + B_1 u_{1t} + C v_{1t}
\]

where
- \( \Lambda_{1t} := A - B_{-1} F_{-1t} \)
- \( \Pi_{1t} := R_t + F'_{-1t} S_t F_{-1t} \)
- \( \Gamma_{1t} := W'_t - M'_t F_{-1t} \)

This is an LQ robust dynamic programming problem of the type studied in the *Robustness* lecture, which can be solved by working backwards.

Maximization with respect to distortion \( v_{1t} \) leads to the following version of the \( D \) operator from the *Robustness* lecture, namely

\[
D_1(P) := P + PC(\theta_1 I - C'PC)^{-1}C'P
\]

The matrix \( F_{1t} \) in the policy rule \( u_{1t} = -F_{1t}x_t \) that solves agent 1's problem satisfies

\[
F_{1t} = (Q_1 + \beta B_1' D_1(P_{1t+1}) B_1)^{-1}(\beta B_1' D_1(P_{1t+1}) \Lambda_{1t} + \Gamma_{1t})
\]

where \( P_{1t} \) solves the matrix Riccati difference equation

\[
P_{1t} = \Pi_{1t} - (\beta B_1' D_1(P_{1t+1}) \Lambda_{1t} + \Gamma_{1t})' (Q_1 + \beta B_1' D_1(P_{1t+1}) B_1)^{-1}(\beta B_1' D_1(P_{1t+1}) \Lambda_{1t} + \Gamma_{1t}) + \beta \Lambda_{1t}' D_1(P_{1t+1}) \Lambda_{1t}
\]

Similarly, the policy that solves player 2's problem is

\[
F_{2t} = (Q_2 + \beta B_2' D_2(P_{2t+1}) B_2)^{-1}(\beta B_2' D_2(P_{2t+1}) \Lambda_{2t} + \Gamma_{2t})
\]
where $P_{2t}$ solves

$$
P_{2t} = \Pi_{2t} - (\beta B_2 D_2 (P_{2t+1}) \Lambda_{2t} + \Gamma_{2t})' (Q_2 + \beta B_2 D_2 (P_{2t+1}) B_2)^{-1} (\beta B_2 D_2 (P_{2t+1}) \Lambda_{2t} + \Gamma_{2t}) + \beta \Lambda_{2t} D_2 (P_{2t+1}) \Lambda_{2t}$$

(7.44)

Here in all cases $t = t_0, \ldots, t_1 - 1$ and the terminal conditions are $P_{it_1} = 0$

The solution procedure is to use equations (7.41), (7.42), (7.43), and (7.44), and work backwards from time $t_1 - 1$

Since were 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 (7.41) contain $F_{2t}$
- some terms on the right hand side of (7.43) contain $F_{1t}$

we need to solve these $k_1 + k_2$ equations simultaneously

**Key insight**

As in Markov perfect equilibrium, a key insight here is that equations (7.41) and (7.43) 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 (7.42) and (7.44)

Notice how $j$s control law $F_{jt}$ is a function of \{F_{is}, s \geq t, i \neq j\}

Thus, agent $i$’s choice of \{F_{it}; t = t_0, \ldots, t_1 - 1\} influences agent $j$s choice of control laws

However, in the Markov perfect equilibrium of this game, each agent is assumed to ignore the influence that his choice exerts on the other agents choice

After these equations have been solved, we can also deduce associated sequences of worst-case shocks

**Worst-case shocks**

For agent $i$ the maximizing or worst-case shock $v_{it}$ is

$$v_{it} = K_{it} x_t$$

where

$$K_{it} = \theta_i^{-1} (I - \theta_i^{-1} C' P_{i,t+1} C)^{-1} C' P_{i,t+1} (A - B_1 F_{it} - B_2 F_{2t})$$

**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 $t_1 \rightarrow +\infty$

In practice, we usually fix $t_1$ and compute the equilibrium of an infinite horizon game by driving $t_0 \rightarrow -\infty$

This is the approach we adopt in the next section

7.5. Robust Markov Perfect Equilibrium
Implementation

We use the function \texttt{nnash\_robust} to compute a Markov perfect equilibrium of the infinite horizon linear quadratic dynamic game with robust planers in the manner described above.

7.5.3 Application

A duopoly model

Without concerns for robustness, the model is identical to the duopoly model from the \textit{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

\begin{equation}
    p = a_0 - a_1 (q_1 + q_2)
\end{equation}

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 (7.45) 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:

\begin{equation}
    \pi_i = pq_i - \gamma (\hat{q}_i - q_i)^2, \quad \gamma > 0,
\end{equation}

Substituting the inverse demand curve (7.45) into (7.46) lets us express the one-period payoff as

\begin{equation}
    \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,
\end{equation}

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 (7.46) 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

\[
A := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B_1 := \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad B_2 := \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}
\]

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:

\[
x_{t+1} = (A - B_1 F_1 - B_2 F_2) x_t
\]

\[(7.48)\]

**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

```python
# @authors: Chase Coleman, Thomas Sargent, John Stachurski

import numpy as np
import quantecon as qe
```

7.5. Robust Markov Perfect Equilibrium
# == Parameters == #
\(a_0 = 10.0\)
\(a_1 = 2.0\)
\(\beta = 0.96\)
\(\gamma = 12.0\)

# == In LQ form == #
\(A = \text{np}.\text{eye}(3)\)
\(B_1 = \text{np}.\text{array}([[0., 1., 0.], [0., 0., 1.], [0., 0., 0.]])\)
\(B_2 = \text{np}.\text{array}([[0., 0., 1.], [0., 1., 0.], [0., 0., 0.]])\)
\(R_1 = \begin{bmatrix}
0, & -a_0 / 2, & 0. \\
-a_0 / 2, & a_1, & a_1 / 2. \\
0, & a_1 / 2, & 0. \\
\end{bmatrix}\)
\(R_2 = \begin{bmatrix}
0, & 0, & -a_0 / 2 \\
0, & 0, & a_1 / 2. \\
-a_0 / 2, & a_1 / 2, & a_1 \\
\end{bmatrix}\)
\(Q_1 = Q_2 = \gamma\)
\(S_1 = S_2 = W_1 = W_2 = M_1 = M_2 = 0.0\)

# == Solve using QE's nnash function == #
\(F_1, F_2, P_1, P_2 = \text{qe}.\text{nnash}(A, B_1, B_2, R_1, R_2, Q_1, Q_2, S_1, S_2, W_1, W_2, M_1, M_2, \beta=\beta, \text{tol}=1e-8, \text{max_iter}=1000)\)

# == Display policies == #
print("Computed policies for firm 1 and firm 2:\n")
print(f"F1 = {F1}\")
print(f"F2 = {F2}\")
print("\n")

**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 \(D(P)\) into the backward induction

The MPE with robustness function is `nnash_robust`

The functions code is as follows

```python
from scipy.linalg import solve
import matplotlib.pyplot as plt

def nnash_robust(A, C, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2, 
                 \(\theta_1, \theta_2, \beta=1.0, \text{tol}=1e-8, \text{max_iter}=1000\):
    r""
    Compute the limit of a Nash linear quadratic dynamic game with 
    robustness concern.
```

858 Chapter 7. Multiple Agent Models
In this problem, player $i$ minimizes
\[
\sum_{t=0}^{\infty} \left\{ x_t' r_i x_t + 2 x_t' w_i u_{it} + u_{it}' q_i u_{it} + u_{jt}' s_i u_{jt} + 2 u_{jt}' m_i u_{it} \right\}
\]
subject to the law of motion
\[
x_{it+1} = A x_t + b_1 u_{1t} + b_2 u_{2t} + C w_{it+1}
\]
and a perceived control law \( u_j(t) = - f_j x_t \) for the other player.

The player $i$ also concerns about the model misspecification, and maximizes
\[
\sum_{t=0}^{\infty} \left\{ \beta^{t+1} \theta_{i} w_{it+1}'w_{it+1} \right\}
\]

The solution computed in this routine is the \( f_i \) and \( P_i \) 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 rate
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

A1 = A - B2 @ F2
A2 = A - B1 @ F1
P1 = P1 - (B1.T @ D1P1 @ A1 + Γ1).T @ F1 + \ A1.T @ D1P1 @ A1
P2 = P2 - (B2.T @ D2P2 @ A2 + Γ2).T @ F2 + \ A2.T @ D2P2 @ A2

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

7.5. Robust Markov Perfect Equilibrium 861
Some details

Firm $i$ wants to minimize

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{ x_t' R_i x_t + u_{it} Q_i u_{it} + u_{it}' S_i u_{it} + 2 x_t' W_i u_{it} + 2 u_{it}' M_i u_{it} \right\}$$

where

$$x_t := \begin{bmatrix} 1 \\ q_{1t} \\ q_{2t} \end{bmatrix} \quad \text{and} \quad u_{it} := q_{i,t+1} - q_i \quad i = 1, 2$$

and

$$R_1 := \begin{bmatrix} 0 & -a_0 & 0 \\ -a_0/2 & a_1 & a_1/2 \\ 0 & a_1/2 & 0 \end{bmatrix}, \quad R_2 := \begin{bmatrix} 0 & 0 & -a_0/2 \\ 0 & 0 & a_1 \\ -a_0/2 & a_1/2 & a_1 \end{bmatrix}, \quad Q_1 = Q_2 = \gamma, \quad S_1 = S_2 = 0, \quad W_1 = W_2 = 0, \quad M_1 = M_2 = 0.0$$

The parameters of the duopoly model are:

- $a_0 = 10$
- $a_1 = 2$
- $\beta = 0.96$
- $\gamma = 12$

```python
# == Parameters == #
a0 = 10.0
a1 = 2.0
beta = 0.96
gamma = 12.0

# == In LQ form == #
A = np.eye(3)
B1 = np.array([[0.0], [1.0], [0.0]])
B2 = np.array([[0.0], [0.0], [1.0]])

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 = gamma
S1 = S2 = W1 = W2 = M1 = M2 = 0.0
```

862 Chapter 7. Multiple Agent Models
Consistency check

We first conduct a comparison test to check if \texttt{nnash\_robust} agrees with \texttt{qe.nnash} in the non-robustness case in which each $\theta_i \approx +\infty$.

```python
# == 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=
# == 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=
```

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 \textbf{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 \textit{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:

- $\theta_1 = 0.02$
- $\theta_2 = 0.04$
- $C = \begin{pmatrix} 0 & 0.01 \\ 0.01 & 0.01 \end{pmatrix}$

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

```python
# == Robustness parameters and matrix == #
C = np.asmatrix([[0], [0.01], [0.01]])
theta_1 = 0.02
theta_2 = 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, theta_1, theta_2, beta=beta)

# == 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 == #
AR = A - B1 @ F1r - B2 @ F2r
xr = np.empty((3, n))
xr[:, 0] = 1, 1, 1
for t in range(n - 1):
    xr[:, t + 1] = AR @ xr[:, t]
qr1 = xr[1, :]
qr2 = xr[2, :]
qr = qr1 + qr2  # Total output, RMPE
pr = a0 - a1 + qr  # Price, RMPE
```
```python
# == 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]

qrp1 = xrp1[1, :]
qrp2 = xrp2[1, :]

qrp11 = qrp1 + qrp12  # Total output, RMPE from player 1's belief
qrp2 = qrp2[1, :]
qrp21 = qrp2[2, :]
qrp22 = qrp2[2, :]

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 a the decision rules under a Markov perfect equilibrium with robust firms with multiplier parameters \( θ_i, i = 1, 2 \) set as described above.

Both industry output and price are under the transition dynamics associated with the baseline model; only the decision rules \( F_i \) differ across the two equilibrium objects presented.

```
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

```python
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 = axes[1]
ax.plot(p1, 'g-', lw=2, alpha=0.75, label='firm 1 MPE price')
ax.plot(pr1, 'b-', lw=2, alpha=0.75, label='firm 1 RMPE price')
ax.set(ylabel='price', xlabel='time')
```
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 firms intertemporal objective)

**Heterogeneous beliefs**

As before, let $A^o = A - B_1 F_1^r - B_2 F_2^r$, 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

- $A^o$
- $A^o + C K_1$
- $A^o + C K_2$

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 associated price and total output sequences

The following code plots them

```python
print('Baseline Robust transition matrix AO is: \\
', np.round(AO, 3))
print('Player 1\'s worst-case transition matrix AOCK1 is: \\
', np.round(AOCK1, 3))
print('Player 2\'s worst-case transition matrix AOCK2 is: \\
', 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]
```
Player 2's worst-case transition matrix AOCK2 is:

\[
\begin{bmatrix}
0.999 & 0. & 0.001 \\
0.665 & 0.683 & -0.073 \\
0.67 & -0.071 & 0.695
\end{bmatrix}
\]

# == Plot == #

```python
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.

([HJS08] discuss how this property of robust decision rules is connected to the concept of admissibility in Bayesian statistical decision theory.)
7.6 Asset Pricing I: Finite State Models

A little knowledge of geometric series goes a long way – Robert E. Lucas, Jr.

Asset pricing is all about covariances – Lars Peter Hansen

7.6.1 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

7.6.2 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 \( d_{t+1}, d_{t+2}, \ldots \)
Lets 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)

**Risk Neutral Pricing**

Our first scenario is risk-neutral pricing.

Let \( \beta = 1/(1 + \rho) \) be an intertemporal discount factor, where \( \rho \) 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

\[
P_t = \beta E_t[d_{t+1} + p_{t+1}]
\]

This is a simple cost equals expected benefit relationship.

Here \( E_t[y] \) denotes the best forecast of \( y \), conditioned on information available at time \( t \).

**Pricing with Random Discount Factor**

What happens if for some reason traders discount payouts differently depending on the state of the world?

Michael Harrison and David Kreps \([HK79]\) and Lars Peter Hansen and Scott Richard \([HR87]\) showed that in quite general settings the price of an ex-dividend asset obeys

\[
P_t = E_t[m_{t+1}(d_{t+1} + p_{t+1})]
\]

for some stochastic discount factor \( m_{t+1} \).

The fixed discount factor \( \beta \) in \((7.49)\) 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.

**Asset Pricing and Covariances**

Recall that, from the definition of a conditional covariance \( \text{cov}_t(x_{t+1}, y_{t+1}) \), we have

\[
E_t(x_{t+1}y_{t+1}) = \text{cov}_t(x_{t+1}, y_{t+1}) + E_t x_{t+1} E_t y_{t+1}
\]

(7.51)

If we apply this definition to the asset pricing equation \((7.50)\) we obtain
\[ p_t = \mathbb{E}_t m_{t+1} \mathbb{E}_t (d_{t+1} + p_{t+1}) + \text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1}) \]  

(7.52)

It is useful to regard equation (7.52) as a generalization of equation (7.49)

- In equation (7.49), the stochastic discount factor \( m_{t+1} = \beta \), a constant
- In equation (7.49), the covariance term \( \text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1}) \) is zero because \( m_{t+1} = \beta \)

Equation (7.52) 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

**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 (7.50) by \( d_t \) to get

\[ v_t = \mathbb{E}_t \left[ m_{t+1} \frac{d_{t+1}}{d_t} (1 + v_{t+1}) \right] \]  

(7.53)

Below we'll discuss the implication of this equation

**7.6.3 Prices in the Risk Neutral Case**

What can we say about price dynamics on the basis of the models described above?

The answer to this question depends on

1. the process we specify for dividends
2. the stochastic discount factor and how it correlates with dividends

For now, let's focus on the risk neutral case, where the stochastic discount factor is constant, and study how prices depend on the dividend process

**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 (7.49) and iterating forward gives

\[ p_t = \beta(d + p_{t+1}) \]
\[ = \beta(d + \beta(d + p_{t+2})) \]
\[ \vdots \]
\[ = \beta(d + \beta d + \beta^2 d + \ldots + \beta^{k-2} d + \beta^{k-1} p_{t+k}) \]
Unless prices explode in the future, this sequence converges to

\[ \bar{p} := \frac{\beta d}{1 - \beta} \]  

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 \( p_t = \beta(d + p_{t+1}) \).

**Example 2: Dividends with deterministic growth paths**

Consider a growing, non-random dividend process \( d_{t+1} = gd_t \) 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 (7.53) 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**.

**Example 3: Markov growth, risk neutral pricing**

Next we consider a dividend process

\[ d_{t+1} = g_{t+1}d_t \]  

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

\[ P(x, y) := P\{X_{t+1} = y \mid X_t = x\}, \quad (x, y \in S) \]

2. \( g \) 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 Tauchens method
- $g_t = \exp(X_t)$, so that $\ln g_t = X_t$ is the growth rate

```python
import numpy as np
import matplotlib.pyplot as plt
import quantecon asqe

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 (7.53) 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

$$v(x) = \beta \sum_{y \in S} K(x,y)(1 + v(y))$$

where $K(x,y) := g(y)P(x,y)$ (7.56)

Suppose that there are $n$ possible states $x_1, \ldots, x_n$. 
We can then think of (7.56) as \( n \) stacked equations, one for each state, and write it in matrix form as

\[
v = \beta K (\mathbf{1} + v)
\]

(7.57)

Here

- \( v \) is understood to be the column vector \((v(x_1), \ldots, v(x_n))'\)
- \( K \) is the matrix \((K(x_i, x_j))_{1 \leq i, j \leq n}\)
- \( \mathbf{1} \) is a column vector of ones

When does (7.57) have a unique solution?

From the Neumann series lemma and Gelfands formula, this will be the case if \( \beta K \) has spectral radius strictly less than one.

In other words, we require that the eigenvalues of \( K \) be strictly less than \( \beta^{-1} \) in modulus.

The solution is then

\[
v = (I - \beta K)^{-1} \beta K \mathbf{1}
\]

(7.58)

**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:

```python
from numpy.linalg import eigvals, solve

n = 25  # size of state space
beta = 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 / beta, warning_message

I = np.identity(n)
v = solve(I - beta * K, beta * 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()
```

7.6. Asset Pricing I: Finite State Models
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.

Anticipation of high future dividend growth leads to a high price-dividend ratio.

### 7.6.4 Asset Prices under Risk Aversion

Now let’s turn to the case where agents are risk averse.

Well 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

#### Pricing a Lucas tree

Let’s start with a version of the celebrated asset pricing model of Robert E. Lucas, Jr. \[Luc78\]

As in \[Luc78\], suppose that the stochastic discount factor takes the form
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 (7.55)

The asset being priced is a claim on the endowment process

Following [Luc78], suppose further that in equilibrium, consumption is equal to the endowment, so that \( d_t = c_t \) for all \( t \)

For utility, well assume the \textbf{constant relative risk aversion} (CRRA) specification

\[
u(c) = \frac{c^{1-\gamma}}{1-\gamma} \text{ with } \gamma > 0 \tag{7.60}\]

When \( \gamma = 1 \) we let \( u(c) = \ln c \)

Inserting the CRRA specification into (7.59) and using \( c_t = d_t \) gives

\[
m_{t+1} = \beta \left( \frac{c_{t+1}}{c_t} \right)^{-\gamma} = \beta g_{t+1}^{-\gamma} \tag{7.61}\]

Substituting this into (7.53) 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

\[
v = \beta J(\kappa + v) \]

Assuming that the spectral radius of \( J \) is strictly less than \( \beta^{-1} \), this equation has the unique solution

\[
v = (I - \beta J)^{-1} \beta \kappa \tag{7.62}\]

We will define a function \texttt{tree\_price} to solve for \( v \) given parameters stored in the class \texttt{AssetPriceModel}
class AssetPriceModel:
    """
    A class that stores the primitives of the asset pricing model.

    Parameters
    ----------
    β : scalar, float
        Discount factor
    mc : MarkovChain
        Contains the transition matrix and set of state values for the state process
    γ : scalar(float)
        Coefficient of risk aversion
    g : callable
        The function mapping states to growth rates
    """
    def __init__(self, β=0.96, mc=None, γ=2.0, g=np.exp):
        self.β, self.γ = β, γ
        self.g = g

        # == A default process for the Markov chain == #
        if mc is None:
            self.ρ = 0.9
            self.σ = 0.02
            self.mc = qe.tauchen(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

```python
"""
# == 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 is a plot of $v$ as a function of the state for several values of $γ$, with a positively correlated Markov process and $g(x) = \exp(x)$.

```python
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=r'$\gamma = \{\gamma\}$')

ax.set_title('Price-dividend ratio as a function of the state')
ax.set_ylabel('price-dividend ratio')
ax.set_xlabel('state')
ax.legend(loc='upper right')
plt.show()
```
Notice that $v$ is decreasing in each case.

This is because, with a positively correlated state process, higher states suggest higher future consumption growth.

In the stochastic discount factor (7.61), 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 $P[i] = e^i$ for all $i$ and applying Neumanns geometric series lemma, we are led to

$$v = \beta(1 - \beta P)^{-1} = \beta \sum_{i=0}^{\infty} \beta^i P[i] = \frac{1}{1 - \beta}\frac{1}{\beta}$$

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 (7.58).

This is as expected, since $\gamma = 0$ implies $u(c) = c$ (and hence agents are risk neutral).
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

- $\zeta$ in period $t + 1$, plus
- the right to sell the claim for $p_{t+1}$ next period

The price satisfies (7.50) with $d_t = \zeta$, or

$$ p_t = \mathbb{E}_t [m_{t+1} (\zeta + p_{t+1})] $$

We maintain the stochastic discount factor (7.61), so this becomes

$$ p_t = \mathbb{E}_t \left[ \beta g_{t+1} (\zeta + p_{t+1}) \right] \quad (7.63) $$

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

$$ p = (I - \beta M)^{-1} \beta M \zeta $$ \quad (7.64)

The above is implemented in the function `consol_price`

```python
def consol_price(ap, \zeta):
    
    Computes price of a consol bond with payoff $\zeta$
    
    Parameters
    ----------
    ap: AssetPriceModel
    An instance of AssetPriceModel containing primitives

    $\zeta$ : scalar(float)
    Coupon of the console

    Returns
    -------
    $p$ : array_like(float)
    Console bond prices
    ```
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

\[
w(X_t, p_S) = \max \left\{ \beta \mathbb{E}_t w'(c_{t+1}) w(X_{t+1}, p_S), p(X_t) - p_S \right\}
\]

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
\[
  w(x, p_S) = \max \left\{ \beta \sum_{y \in S} P(x, y) g(y)^{-\gamma} w(y, p_S), \ p(x) - p_S \right\}
\]  

(7.65)

With \( M(x, y) = P(x, y) g(y)^{-\gamma} \) and \( w \) as the vector of values \( (w(x_i), p_S)_i \), we can express (7.65) as the nonlinear vector equation

\[
  w = \max \{ \beta M w; p - p_S \}
\]  

(7.66)

To solve (7.66), form the operator \( T \) mapping vector \( w \) into vector \( Tw \) via

\[
  Tw = \max \{ \beta M w; p - p_S \}
\]

Start at some initial \( w \) and iterate to convergence with \( T \).

We can find the solution with the following function \textit{call_option}

```python
def call_option(ap, \zeta, p_s, =1e-7):
    """
    Computes price of a call option on a consol bond.

    Parameters
    ----------
    ap: AssetPriceModel
        An instance of AssetPriceModel containing primitives
    \zeta : scalar(float)
        Coupon of the console
    p_s : scalar(float)
        Strike price
    
    Returns
    -------
    w : array_like(float)
        Infinite horizon call option prices
    """
    # == Simplify names, set up matrices == #
    \beta, \gamma, P, y = ap.\beta, ap.\gamma, ap.mc.P, ap.mc.state_values
    M = P * ap.g(y)**(-\gamma)

    # == Make sure that a unique consol price exists == #
    ap.test_stability(M)

    # == Compute option price == #
    p = consol_price(ap, \zeta)
```

7.6. Asset Pricing I: Finite State Models
w = np.zeros(ap.n)
error = +1
while error > 0:
    # == 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 $P_S = 40$

```python
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 is high will eventually be visited.

The reason is that $\beta = 0.9$, so the future is discounted relatively rapidly.

**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

$$E_t m_{t+1} = \beta \sum_{y \in S} P(x,y) g(y)^{-\gamma}$$

We can write this as

$$m_1 = \beta M \kappa$$

where the $i$-th element of $m_1$ is the reciprocal of the one-period gross risk-free interest rate in state $x_i$. 

7.6. Asset Pricing I: Finite State Models
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 \( m_{j+1} = Mm_j \) for \( j \geq 1 \)

7.6.5 Exercises

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 (7.49), find the risk-neutral asset pricing equation for one unit of a cum-dividend asset

With a constant, non-random dividend stream \( d_t = d > 0 \), what is the equilibrium price of a cum-dividend asset?

With a growing, non-random dividend process \( d_t = gd_t \) where \( 0 < g/\beta < 1 \), what is the equilibrium price of a cum-dividend asset?

Exercise 2

Consider the following primitives

```python
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 of the Markov chain
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 \( p_S = 150.0 \)

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 (7.65)

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

$$w_k = \max \{ \beta M w_{k-1}, p - p_S \} \quad k = 1, 2, \ldots \quad \text{with } w_0 = 0$$

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?

### 7.6.6 Solutions

#### 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

$$p_t = \frac{1}{1 - \beta g} d_t$$

#### Exercise 2

First let's enter the parameters:

```python
n = 5
P = 0.0125 * np.ones((n, n))
P += np.diag(0.95 - 0.0125 * np.ones(n))
s = np.array([0.95, 0.975, 1.0, 1.025, 1.05])  # state values
mc = qe.MarkovChain(P, state_values=s)
```

7.6. Asset Pricing I: Finite State Models
\[ \gamma = 2.0 \]
\[ \beta = 0.94 \]
\[ \zeta = 1.0 \]
\[ p_s = 150.0 \]

Next well create an instance of `AssetPriceModel` to feed into the functions.

```python
apm = AssetPriceModel(\beta=\beta, mc=mc, \gamma=\gamma, g=\text{lamba } x: x)
```

Now we just need to call the relevant functions on the data:

```python
tree_price(apm)
array([ 29.47401578, 21.93570661, 17.57142236, 14.72515002, 12.72221763])
```

```python
consol_price(apm, \zeta)
array([ 753.87100476, 242.55144082, 148.67554548, 109.25108965, 87.56860139])
```

```python
call_option(apm, \zeta, p_s)
array([ 603.87100476, 176.83933430, 108.67734499, 80.05179254, 64.30843748])
```

Lets show the last two functions as a plot.

```python
fig, ax = plt.subplots()
ax.plot(s, consol_price(apm, \zeta), label='consol')
ax.plot(s, call_option(apm, \zeta, p_s), label='call option')
ax.legend()
plt.show()
```
Exercise 3

Here is a suitable function:

```python
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 $k=25$.

```python
fig, ax = plt.subplots()
for k in [5, 25]:
    w = finite_horizon_call_option(apm, ζ, p_s, k)
```

7.6. Asset Pricing I: Finite State Models
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.

### 7.7 Asset Pricing II: The Lucas Asset Pricing Model

**Contents**

- Asset Pricing II: The Lucas Asset Pricing Model
  - Overview
  - The Lucas Model
  - Exercises
  - Solutions

#### 7.7.1 Overview

As stated in an earlier lecture, an asset is a claim on a stream of prospective payments.

What is the correct price to pay for such a claim?
The elegant asset pricing model of Lucas [Luc78] attempts to answer this question in an equilibrium setting with risk averse agents.

While we mentioned some consequences of Lucas model earlier, it is now time to work through the model more carefully, and try to understand where the fundamental asset pricing equation comes from.

A side benefit of studying Lucas model is that it provides a beautiful illustration of model building in general and equilibrium pricing in competitive models in particular.

Another difference to our first asset pricing lecture is that the state space and shock will be continuous rather than discrete.

### 7.7.2 The Lucas Model

Lucas studied a pure exchange economy with a representative consumer (or household), where

- **Pure exchange** means that all endowments are exogenous
- **Representative** consumer means that either
  - there is a single consumer (sometimes also referred to as a household), or
  - all consumers have identical endowments and preferences

Either way, the assumption of a representative agent means that prices adjust to eradicate desires to trade. This makes it very easy to compute competitive equilibrium prices.

### Basic Setup

Let's review the set up.

### Assets

There is a single productive unit that costlessly generates a sequence of consumption goods \( \{y_t\}_{t=0}^{\infty} \). Another way to view \( \{y_t\}_{t=0}^{\infty} \) is as a consumption endowment for this economy.

We will assume that this endowment is Markovian, following the exogenous process

\[
y_{t+1} = G(y_t, \xi_{t+1})
\]

Here \( \{\xi_t\} \) is an iid shock sequence with known distribution \( \phi \) and \( y_t \geq 0 \).

An asset is a claim on all or part of this endowment stream.

The consumption goods \( \{y_t\}_{t=0}^{\infty} \) are nonstorable, so holding assets is the only way to transfer wealth into the future.

For the purposes of intuition, it's common to think of the productive unit as a tree that produces fruit.

Based on this idea, a Lucas tree is a claim on the consumption endowment.
Consumers

A representative consumer ranks consumption streams \( \{c_t\} \) according to the time separable utility functional

\[
E \sum_{t=0}^{\infty} \beta^t u(c_t)
\] (7.67)

Here

- \( \beta \in (0, 1) \) is a fixed discount factor
- \( u \) is a strictly increasing, strictly concave, continuously differentiable period utility function
- \( E \) is a mathematical expectation

Pricing a Lucas Tree

What is an appropriate price for a claim on the consumption endowment?

Well price an \textit{ex dividend} claim, meaning that

- the seller retains this periods 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 well impose equilibrium constraints and try to back out prices

In the consumer problem, the consumers control variable is the share \( \pi_t \) of the claim held in each period

Thus, the consumer problem is to maximize (7.67) subject to

\[
c_t + \pi_{t+1} p_t \leq \pi_t y_t + \pi_t p_t
\]

along with \( c_t \geq 0 \) and \( 0 \leq \pi_t \leq 1 \) at each \( t \)

The decision to hold share \( \pi_t \) is actually made at time \( t - 1 \)

But this value is inherited as a state variable at time \( t \), which explains the choice of subscript

The dynamic program

We can write the consumer problem as a dynamic programming problem

Our first observation is that prices depend on current information, and current information is really just the endowment process up until the current period
In fact the endowment process is Markovian, so that the only relevant information is the current state $y \in \mathbb{R}_+$ (dropping the time subscript).

This leads us to guess an equilibrium where price is a function $p$ of $y$.

Remarks on the solution method

- Since this is a competitive (read: price taking) model, the consumer will take this function $p$ as given.
- In this way we determine consumer behavior given $p$ and then use equilibrium conditions to recover $p$.
- This is the standard way to solve competitive equilibrium models.

Using the assumption that price is a given function $p$ of $y$, we write the value function and constraint as

$$v(\pi, y) = \max_{c, \pi'} \left\{ u(c) + \beta \int v(\pi', G(y, z)) \phi(dz) \right\}$$

subject to

$$c + \pi' p(y) \leq \pi y + \pi p(y)$$

(7.68)

We can invoke the fact that utility is increasing to claim equality in (7.68) and hence eliminate the constraint, obtaining

$$v(\pi, y) = \max_{\pi'} \left\{ u[\pi(y + p(y)) - \pi' p(y)] + \beta \int v(\pi', G(y, z)) \phi(dz) \right\}$$

(7.69)

The solution to this dynamic programming problem is an optimal policy expressing either $\pi'$ or $c$ as a function of the state $(\pi, y)$.

- Each one determines the other, since $c(\pi, y) = \pi(y + p(y)) - \pi'(\pi, y)p(y)$.

Next steps

What we need to do now is determine equilibrium prices.

It seems that to obtain these, we will have to

1. Solve this two dimensional dynamic programming problem for the optimal policy.
2. Impose equilibrium constraints.
3. Solve out for the price function $p(y)$ directly.

However, as Lucas showed, there is a related but more straightforward way to do this.
Equilibrium constraints

Since the consumption good is not storable, in equilibrium we must have \( c_t = y_t \) for all \( t \).

In addition, since there is one representative consumer (alternatively, since all consumers are identical), there should be no trade in equilibrium.

In particular, the representative consumer owns the whole tree in every period, so \( \pi_t = 1 \) for all \( t \).

Prices must adjust to satisfy these two constraints.

The equilibrium price function

Now observe that the first order condition for (7.69) 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 (7.69) with respect to \( \pi \), yielding

\[
  v'_1(\pi, y) = u'(c)(y + p(y))
\]

Next we impose the equilibrium constraints while combining the last two equations to get

\[
  p(y) = \beta \int \frac{u'(G(y, z))}{u'(y)}[G(y, z) + p(G(y, z))]\phi(dz) \tag{7.70}
\]

In sequential rather than functional notation, we can also write this as

\[
  p_t = \mathbb{E}_t \left[ \beta \frac{u'(c_{t+1})}{u'(c_t)}(y_{t+1} + p_{t+1}) \right] \tag{7.71}
\]

This is the famous consumption-based asset pricing equation.

Before discussing it further we want to solve out for prices.

Solving the Model

Equation (7.70) 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 well follow Lucas indirect approach, first setting

\[ f(y) := u'(y)p(y) \quad (7.72) \]

so that (7.70) becomes

\[ f(y) = h(y) + \beta \int f(G(y, z)]\phi(dz) \quad (7.73) \]

Here \( h(y) := \beta \int u'[G(y, z)]G(y, z)\phi(dz) \) is a function that depends only on the primitives.

Equation (7.73) is a functional equation in \( f \).

The plan is to solve out for \( f \) and convert back to \( p \) via (7.72).

To solve (7.73) well use a standard method: convert it to a fixed point problem.

First we introduce the operator \( T \) mapping \( f \) into \( Tf \) as defined by

\[ (Tf)(y) = h(y) + \beta \int f[G(y, z)]\phi(dz) \quad (7.74) \]

The reason we do this is that a solution to (7.73) 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 \( f: \mathbb{R}_+ \rightarrow \mathbb{R}_+ \).

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

\[ \|Tf - Tg\| \leq \alpha\|f - g\|, \quad \forall f, g \in cb\mathbb{R}_+ \quad (7.75) \]
Here \( \|h\| := \sup_{x \in \mathbb{R}_+} |h(x)| \)

To see that (7.75) 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,

\[
|Tf(y) - Tg(y)| = \left| \beta \int f(G(y, z)) \phi(dz) - \beta \int g(G(y, z)) \phi(dz) \right| \\
\leq \beta \int |f(G(y, z)) - g(G(y, z))| \phi(dz) \\
\leq \beta \int \|f - g\| \phi(dz) \\
= \beta \|f - g\|
\]

Since the right hand side is an upper bound, taking the sup over all \( y \) on the left hand side gives (7.75) with \( \alpha := \beta \)

**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 \( u(c) = c^{\gamma - 1}/(1 - \gamma) \), where \( \gamma > 0 \) is the coefficient of relative risk aversion

Some code to implement the iterative computational procedure can be found in lucastree.py

We repeat it here for convenience

```python
import numpy as np
from scipy.stats import lognorm
from scipy.integrate import fixed_quad

class LucasTree:
    ""
    Class to store parameters of a the Lucas tree model, a grid for the iteration step and some other helpful bits and pieces.
    ""
    Parameters
    ----------
    \gamma : scalar(float)
        The coefficient of risk aversion in the household's CRRA utility function
    \beta : scalar(float)
        The household's discount factor
    \alpha : scalar(float)
        The correlation coefficient in the shock process
    \sigma : scalar(float)
```

898 Chapter 7. Multiple Agent Models
The volatility of the shock process
grid_size : int
The size of the grid to use

Attributes
---------
\( \gamma, \beta, \alpha, \sigma \), grid_size : see Parameters
grid : ndarray
   Properties for grid upon which prices are evaluated
   : scipy.stats.lognorm
   The distribution for the shock process

Examples
--------
```python
def __init__(self, 
    \( \gamma=2 \), 
    \( \beta=0.95 \), 
    \( \alpha=0.90 \), 
    \( \sigma=0.1 \), 
    grid_size=100):

    self.\( \gamma \), self.\( \beta \), self.\( \alpha \), self.\( \sigma \) = \( \gamma \), \( \beta \), \( \alpha \), \( \sigma \)

    # == Set the grid interval to contain most of the mass of the
    # stationary distribution of the consumption endowment == #
    ssd = self.\( \sigma \) / np.sqrt(1 - self.\( \alpha \)^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. = lognorm(\( \sigma \))
    self.draws = self..rvs(500)

    # == h(y) = \( \beta \) * int G(y,z)^(1-\( \gamma \)) (dz) == #
    self.h = np.empty(self.grid_size)
    for i, y in enumerate(self.grid):
        self.h[i] = \( \beta \) * np.mean((y\( \alpha \) * self.draws)^\( 1 - \gamma \))

# == Now the functions that act on a Lucas Tree == #
def lucas_operator(f, tree, Tf=None):
    
    The approximate Lucas operator, which computes and returns the
    updated function Tf on the grid points.
Parameters
----------
f : array_like(float)
   A candidate function on R_+ represented as points on a grid
   and should be flat NumPy array with len(f) = len(grid)

.tree : instance of LucasTree
   Stores the parameters of the problem

Tf : array_like(float)
   Optional storage array for Tf

Returns
-------
Tf : array_like(float)
   The updated function Tf

Notes
-----
The argument `Tf` is optional, but recommended. If it is passed
into this function, then we do not have to allocate any memory
for the array here. As this function is often called many times
in an iterative algorithm, this can save significant computation
time.

```python
grid, h = tree.grid, tree.h
α, β = tree.α, tree.β
z_vec = tree.draws

# == turn f into a function == #
Af = lambda x: np.interp(x, grid, f)

# == set up storage if needed ==#
if Tf is None:
    Tf = np.empty_like(f)

# == Apply the T operator to f using Monte Carlo integration ==#
for i, y in enumerate(grid):
    Tf[i] = h[i] + β * np.mean(Af(y**α * z_vec))

return Tf
```

def solve_lucas_model(tree, tol=1e-6, max_iter=500):
    ""
    Compute the equilibrium price function associated with Lucas
tree
    ""
    Parameters
    ----------
    tree : An instance of LucasTree
        Contains parameters
tol : float
error tolerance
max_iter : int
    the maximum number of iterations

Returns
-------
price : array_like(float)
    The prices at the grid points in the attribute `grid` of the object

# == simplify notation == #
grid, grid_size = tree.grid, tree.grid_size
γ = tree.γ

# == Create storage array for lucas_operator. Reduces memory allocation and speeds code up == #
Tf = np.empty(grid_size)

i = 0
f = np.empty(grid_size)  # Initial guess of f
error = tol + 1

while error > tol and i < max_iter:
    f_new = lucas_operator(f, tree, Tf)
    error = np.max(np.abs(f_new - f))
    f[:] = f_new
    i += 1

price = f * grid**γ  # Back out price vector

return price

An example of usage is given in the docstring and repeated here

tree = LucasTree()
price_vals = solve_lucas_model(tree)

Here's the resulting price function

```python
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 8))
plt.plot(tree.grid, price_vals, label='$p*(y)$')
plt.xlabel('$y$')
plt.ylabel('price')
plt.legend()
plt.show()
```
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.

### 7.7.3 Exercises

**Exercise 1**

Replicate the figure to show how discount rates affect prices.

### 7.7.4 Solutions

```python
import matplotlib.pyplot as plt
```

**Exercise 1**

Note that this code assumes you have run the lucastree.py script embedded above.
7.8 Asset Pricing III: Incomplete Markets
7.8.1 Overview

This lecture describes a version of a model of Harrison and Kreps [HK78]

The model determines the price of a dividend-yielding asset that is traded by two types of self-interested investors

The model features

- heterogeneous beliefs
- incomplete markets
- short sales constraints, and possibly ...
- (leverage) limits on an investor's ability to borrow in order to finance purchases of a risky asset

References

Prior to reading the following you might like to review our lectures on

- Markov chains
- Asset pricing with finite state space

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
7.8.2 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

$$P_a = \begin{bmatrix} \frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

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:

```python
import numpy as np
import quantecon as qe

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
mcB.stationary_distributions
```

array([[ 0.57142857, 0.42857143]])

```
mcB.stationary_distributions
```

array([[ 0.42857143, 0.57142857]])

The stationary distribution of $P_a$ is approximately $\pi_A = [0.57, 0.43]$.

The stationary distribution of $P_b$ is approximately $\pi_B = [0.43, 0.57]$. 

906 Chapter 7. Multiple Agent Models
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, well set $\beta = .75$, just as Harrison and Kreps did

Well 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\(^1\)

2. No single type of investor has sufficient resources to purchase the entire stock

Case 1 is the case studied in Harrison and Kreps

In case 2, both types of investor always hold at least some of the asset

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

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 periods dividend than a type $b$ agent
- In state 1, a type $b$ agent is more optimistic about next periods dividend

However, the stationary distributions $\pi_A = [.57 \ .43]$ and $\pi_B = [.43 \ .57]$ 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

---

\(^1\) By assuming that both types of agent 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.
Temporarily optimistic investors (i.e., the investor with the most optimistic beliefs in each state) believe the transition matrix

$$P_o = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

Temporarily pessimistic believe the transition matrix

$$P_p = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

Well return to these matrices and their significance in the exercise

**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 $s_t$

### 7.8.3 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 agent differentiated only by their beliefs. Each type of agent has sufficient resources to purchase all of the asset (Harrison and Krepss setting)
3. There are two types of agent with different beliefs, but because of limited wealth and/or limited leverage, both types of investors hold the asset each period

**Summary Table**

The following table gives a summary of the findings obtained in the remainder of the lecture (you will be asked to recreate the table in an exercise)

It records implications of Harrison and Krepss specifications of $P_a, P_b, \beta$
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

Well explain these values and how they are calculated one row at a time

### Single Belief Prices

Well 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 tomorrows dividend and tomorrows price of the asset:

\[
p_h(s) = \beta (P_h(s,0)(0 + p_h(0)) + P_h(s,1)(1 + p_h(1))) \quad s = 0, 1
\]

These equations imply that the equilibrium price vector is

\[
\begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix} = \beta[I - \beta P_h]^{-1}P_h \begin{bmatrix} 0 \\ 1 \end{bmatrix}
\]

(7.76)

The first two rows of of the table report \( p_a(s) \) and \( p_b(s) \)

Here a function that can be used to compute these values

<table>
<thead>
<tr>
<th>( s_t )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_a )</td>
<td>1.33</td>
<td>1.22</td>
</tr>
<tr>
<td>( p_b )</td>
<td>1.45</td>
<td>1.91</td>
</tr>
<tr>
<td>( p_o )</td>
<td>1.85</td>
<td>2.08</td>
</tr>
<tr>
<td>( p_p )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \hat{p}_a )</td>
<td>1.85</td>
<td>1.69</td>
</tr>
<tr>
<td>( \hat{p}_b )</td>
<td>1.69</td>
<td>2.08</td>
</tr>
</tbody>
</table>
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

Pricing under Heterogeneous Beliefs

There are several cases to consider

The first is when both types of agent 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 key equation:

\[
\bar{p}(s) = \beta \max \{ P_a(s, 0) \bar{p}(0) + P_a(s, 1)(1 + \bar{p}(1)), P_b(s, 0) \bar{p}(0) + P_b(s, 1)(1 + \bar{p}(1)) \} \tag{7.77}
\]

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 (7.77) 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 (7.77), namely

$$
\bar{p}^{j+1}(s) = \beta \max \left\{ P_a(s, 0)\bar{p}^j(0) + P_a(s, 1)(1 + \bar{p}^j(1)), \ P_b(s, 0)\bar{p}^j(0) + P_b(s, 1)(1 + \bar{p}^j(1)) \right\}
$$

(7.78)

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 dont 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

```python
def price_optimistic_beliefs(transitions, dividend_payoff, beta=.75, max_iter=50000, tol=1e-16):
    
    ""
    Function to Solve Optimistic Beliefs
    ""
    # We will guess an initial price vector of [0, 0]
```

7.8. Asset Pricing III: Incomplete Markets
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

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 (7.77), the equilibrium price satisfies

\[
p(s) = \min \{ P_a(s, 1)p(0) + P_a(s, 1)(1 + \hat{p}(1)),
            P_b(s, 1)p(0) + P_b(s, 1)(1 + \hat{p}(1)) \} \tag{7.79}
\]

and the marginal investor who prices the asset is always the one that values it less highly than does the other type

Now the marginal investor is always the (temporarily) pessimistic type

Notice from the sixth row of that the pessimistic price \( p \) is lower than the homogeneous belief prices \( p_a \) and \( p_b \) in both states

When pessimistic investors price the asset according to (7.79), optimistic investors think that the asset is underpriced

If they could, optimistic investors would willingly borrow at the one-period gross interest rate \( \beta^{-1} \) to purchase more of the asset

Implicit constraints on leverage prohibit them from doing so

When optimistic investors price the asset as in equation (7.77), 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 $\tilde{p}$ using iteration

```python
def price_pessimistic_beliefs(transitions, dividend_payoff, $\beta=$.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 = $\beta$ * 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
```

Further Interpretation

[Sch14] interprets the Harrison-Kreps model as a model of a bubble—a situation in which an asset price exceeds what every investor thinks is merited by the assets 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 effects of financial regulations on bubbles

He emphasizes how limiting short sales and limiting leverage have opposite effects

7.8. Asset Pricing III: Incomplete Markets 913
7.8.4 Exercises

Exercise 1

Recreate the summary table using the functions we have built above

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_t$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$p_a$</td>
<td>1.33</td>
<td>1.22</td>
</tr>
<tr>
<td>$p_b$</td>
<td>1.45</td>
<td>1.91</td>
</tr>
<tr>
<td>$p_o$</td>
<td>1.85</td>
<td>2.08</td>
</tr>
<tr>
<td>$p_1$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\hat{p}_a$</td>
<td>1.85</td>
<td>1.69</td>
</tr>
<tr>
<td>$\hat{p}_b$</td>
<td>1.69</td>
<td>2.08</td>
</tr>
</tbody>
</table>

You will first need to define the transition matrices and dividend payoff vector

7.8.5 Solutions

Exercise 1

First we will obtain equilibrium price vectors with homogeneous beliefs, including when all investors are optimistic or pessimistic

```python
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
qopt = np.array([[1/2, 1/2], [1/4, 3/4]]) # Optimistic investor transition matrix
qpess = np.array([[2/3, 1/3], [1/4, 3/4]]) # Pessimistic investor transition matrix

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)
```

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>State 0:</td>
<td>[ 1.33]</td>
<td></td>
</tr>
<tr>
<td>State 1:</td>
<td>[ 1.22]</td>
<td></td>
</tr>
</tbody>
</table>
```

p_a

```
--------------------
State 0:  [ 1.33]
State 1:  [ 1.22]
--------------------
```
We will use the `price_optimistic_beliefs` function to find the price under heterogeneous beliefs

```python
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)
```

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
7.9 Uncertainty Traps

7.9.1 Overview

In this lecture we study a simplified version of an uncertainty traps model of Fajgelbaum, Schaal and Taschereau-Dumouchel [FSTD15]. The model features self-reinforcing uncertainty that has big impacts on economic activity.

In the model,

- Fundamentals vary stochastically and are not fully observable
- At any moment there are both active and inactive entrepreneurs; only active entrepreneurs produce
- Agents – active and inactive entrepreneurs – have beliefs about the fundamentals expressed as probability distributions
- Greater uncertainty means greater dispersions of these distributions
- Entrepreneurs are risk averse and hence less inclined to be active when uncertainty is high
- The output of active entrepreneurs is observable, supplying a noisy signal that helps everyone inside the model infer fundamentals
- Entrepreneurs update their beliefs about fundamentals using Bayes Law, implemented via Kalman filtering

Uncertainty traps emerge because:

- High uncertainty discourages entrepreneurs from becoming active
- A low level of participation – i.e., a smaller number of active entrepreneurs – diminishes the flow of information about fundamentals
- Less information translates to higher uncertainty, further discouraging entrepreneurs from choosing to be active, and so on

Uncertainty traps stem from a positive externality: high aggregate economic activity levels generates valuable information.

7.9.2 The Model

The original model described in [FSTD15] has many interesting moving parts.

Here we examine a simplified version that nonetheless captures many of the key ideas.

Fundamentals

The evolution of the fundamental process \( \{ \theta_t \} \) is given by

\[
\theta_{t+1} = \rho \theta_t + \sigma \omega_{t+1}
\]

where
• $\sigma_\theta > 0$ and $0 < \rho < 1$
• $\{w_t\}$ is IID and standard normal

The random variable $\theta_t$ is not observable at any time

**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

$$x_m = \theta + \epsilon_m \quad \text{where} \quad \epsilon_m \sim N(0, \gamma_x^{-1})$$

(7.80)

Here the time subscript has been dropped to simplify notation

The inverse of the shock variance, $\gamma_x$, is called the shocks precision

The higher is the precision, the more informative $x_m$ is about the fundamental

Output shocks are independent across time and firms

**Information and Beliefs**

All entrepreneurs start with identical beliefs about $\theta_0$

Signals are publicly observable and hence all agents have identical beliefs always

Dropping time subscripts, beliefs for current $\theta$ are represented by the normal distribution $N(\mu, \gamma^{-1})$

Here $\gamma$ is the precision of beliefs; its inverse is the degree of uncertainty

These parameters are updated by Kalman filtering

Let

• $\mathbb{M} \subset \{1, \ldots, \bar{M}\}$ denote the set of currently active firms
• $M := |\mathbb{M}|$ denote the number of currently active firms
• $X$ be the average output $\frac{1}{M} \sum_{m \in \mathbb{M}} x_m$ of the active firms

With this notation and primes for next period values, we can write the updating of the mean and precision via

$$\mu' = \rho \frac{\gamma \mu + M \gamma_x X}{\gamma + M \gamma_x}$$

(7.81)

$$\gamma' = \left(\frac{\rho^2 \gamma + \gamma_x}{\gamma + M \gamma_x + \sigma_\theta^2}\right)^{-1}$$

(7.82)
These are standard Kalman filtering results applied to the current setting.

Exercise 1 provides more details on how (7.81) and (7.82) are derived, and then asks you to fill in remaining steps.

The next figure plots the law of motion for the precision in (7.82) as a 45 degree diagram, with one curve for each $M \in \{0, \ldots, 6\}$.

The other parameter values are $\rho = 0.99, \gamma_x = 0.5, \sigma_\theta = 0.5$.

Points where the curves hit the 45 degree lines are long run steady states for precision for different values of $M$.

Thus, if one of these values for $M$ remains fixed, a corresponding steady state is the equilibrium level of precision.
• high values of $M$ correspond to greater information about the fundamental, and hence more precision in steady state

• low values of $M$ correspond to less information and more uncertainty in steady state

In practice, as well see, the number of active firms fluctuates stochastically

**Participation**

Omitting time subscripts once more, entrepreneurs enter the market in the current period if

$$
E[u(x_m - F_m)] > c
$$

(7.83)

Here

• the mathematical expectation of $x_m$ is based on (7.80) and beliefs $N(\mu, \gamma^{-1})$ for $\theta$

• $F_m$ is a stochastic but previsible fixed cost, independent across time and firms

• $c$ is a constant reflecting opportunity costs

The statement that $F_m$ is previsible means that it is realized at the start of the period and treated as a constant in (7.83)

The utility function has the constant absolute risk aversion form

$$
u(x) = \frac{1}{a} (1 - \exp(-ax))
$$

(7.84)

where $a$ is a positive parameter

Combining (7.83) and (7.84), entrepreneur $m$ participates in the market (or is said to be active) when

$$
\frac{1}{a} \left\{ 1 - E[\exp \left( -a(\theta + \epsilon_m - F_m) \right)] \right\} > c
$$

Using standard formulas for expectations of lognormal random variables, this is equivalent to the condition

$$
\psi(\mu, \gamma, F_m) := \frac{1}{a} \left( 1 - \exp \left( -a\mu + aF_m + \frac{a^2 \left( \frac{1}{\gamma} + \frac{1}{\gamma_\theta} \right)}{2} \right) \right) - c > 0
$$

(7.85)

**7.9.3 Implementation**

We want to simulate this economy

As a first step, let's put together a class that bundles

• the parameters, the current value of $\theta$ and the current values of the two belief parameters $\mu$ and $\gamma$
• methods to update $\theta$, $\mu$ and $\gamma$, as well as to determine the number of active firms and their outputs

The updating methods follow the laws of motion for $\theta$, $\mu$ and $\gamma$ given above

The method to evaluate the number of active firms generates $F_1, \ldots, F_M$ and tests condition (7.85) for each firm

The $\text{__init__}$ method encodes as default values the parameters well use in the simulations below

```python
import numpy as np

class UncertaintyTrapEcon:
    def __init__(self,
        a=1.5,      # Risk aversion
        $\gamma_x$=0.5,  # Production shock precision
        $\rho$=0.99,   # Correlation coefficient for $\theta$
        $\sigma_{\theta}$=0.5,  # Standard dev of $\theta$ shock
        num_firms=100,  # Number of firms
        $\sigma_F$=1.5,  # Standard dev of fixed costs
        c=-420,        # External opportunity cost
        $\mu_{\text{init}}$=0,  # Initial value for $\mu$
        $\gamma_{\text{init}}$=4,  # Initial value for $\gamma$
        $\theta_{\text{init}}$=0):  # Initial value for $\theta$

        # == Record values == #
        self.a, self.$\gamma_x$, self.$\rho$, self.$\sigma_\theta$ = a, $\gamma_x$, $\rho$, $\sigma_\theta$
        self.num_firms, self.$\sigma_F$, self.c, = num_firms, $\sigma_F$, c
        self.$\gamma_x$ = np.sqrt(1/$\gamma_x$)

        # == Initialize states == #
        self.$\gamma$, self.$\mu$, self.$\theta$ = $\gamma_{\text{init}}$, $\mu_{\text{init}}$, $\theta_{\text{init}}$

    def $\psi$(self, F):
        temp1 = -self.a * (self.$\mu$ - F)
        temp2 = self.a*$\gamma_x$ + (1/self.$\gamma$ + 1/self.$\gamma_x$) / 2
        return (1 / self.a) * (1 - np.exp(temp1 + temp2)) - self.c

    def update_beliefs(self, X, M):
        # == Update beliefs == #
        $\gamma_x$, $\rho$, $\sigma_\theta$ = self.$\gamma_x$, self.$\rho$, self.$\sigma_\theta$
        $\mu$ = temp1 / temp2
        self.$\gamma$ = 1 / ($\rho$$\gamma_x$ + M * $\gamma_x$) + $\sigma_\theta$$\gamma_x$
        $\theta$ = self.$\gamma$ + M * $\gamma_x$ + self.$\theta$**2

    def update_\theta(self, w):
```

Chapter 7. Multiple Agent Models
Update the fundamental state $\theta$ given shock $w$.

```python
    self.\theta = self.\rho * self.\theta + self.\sigma_{\theta} * w
```

```python
def gen_aggregates(self):
    """
    Generate aggregates based on current beliefs ($\mu$, $\gamma$). This
    is a simulation step that depends on the draws for $F$.
    """
    F_vals = self.\sigma_F * np.random.randn(self.num_firms)
    M = np.sum(self.\psi(F_vals) > 0)  # Counts number of active firms
    if M > 0:
        x_vals = self.\theta + self.\sigma_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

### 7.9.4 Results

Lets look first at the dynamics of $\mu$, which the agents use to track $\theta$

![Graph showing time series of $\theta$ and $\mu$](image)

We see that $\mu$ tracks $\theta$ well when there are sufficient firms in the market

#### 7.9. Uncertainty Traps
However, there are times when $\mu$ tracks $\theta$ 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.
7.9. Uncertainty Traps
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

7.9.5 Exercises

Exercise 1

Fill in the details behind (7.81) and (7.82) based on the following standard result (see, e.g., p. 24 of [YS05])

Fact Let \( x = (x_1, \ldots, x_M) \) be a vector of IID draws from common distribution \( N(\theta, 1/\gamma_x) \) and let \( \bar{x} \) be the sample mean. If \( \gamma_x \) is known and the prior for \( \theta \) is \( N(\mu, 1/\gamma) \), then the posterior distribution of \( \theta \) given \( x \) is

\[
\pi(\theta | x) = N(\mu_0, 1/\gamma_0)
\]

where

\[
\mu_0 = \frac{\mu \gamma + M \bar{x} \gamma_x}{\gamma + M \gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M \gamma_x
\]

Exercise 2

Modulo randomness, replicate the simulation figures shown above

- Use the parameter values listed as defaults in the \texttt{__init__} method of the \texttt{UncertaintyTrapEcon} class

7.9.6 Solutions

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

Exercise 1

This exercise asked you to validate the laws of motion for \( \gamma \) and \( \mu \) 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 \( \theta \) 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 \( \mu' \) and \( \gamma' \) given in the lecture.
Exercise 2

First let’s replicate the plot that illustrates the law of motion for precision, which is

\[ \gamma_{t+1} = \left( \frac{\rho^2}{\gamma_t + M \gamma_x + \sigma^2} \right)^{-1} \]

Here \( M \) is the number of active firms. The next figure plots \( \gamma_{t+1} \) against \( \gamma_t \) on a 45 degree diagram for different values of \( M \).

```python
econ = UncertaintyTrapEcon()
\rho, \sigma_\theta, \gamma_x = econ.\rho, econ.\sigma_\theta, econ.\gamma_x  # simplify names
\gamma = np.linspace(1e-10, 3, 200)  # \gamma grid
fig, ax = plt.subplots(figsize=(9, 9))  # 45 degree line
ax.plot(\gamma, \gamma, 'k-')

for M in range(7):
    \gamma_next = 1 / (\rho**2 / (\gamma + M * \gamma_x + \sigma_\theta**2))
    label_string = f"$M = {M}$"
    ax.plot(\gamma, \gamma_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 lets generate time series for beliefs and the aggregates – that is, the number of active firms and average output.

```python
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)
```
\[ \mu_{\text{vec}[0]} = \text{econ.} \mu \]
\[ \gamma_{\text{vec}[0]} = \text{econ.} \gamma \]
\[ \theta_{\text{vec}[0]} = 0 \]

\[ w_{\text{shocks}} = \text{np.random.randn(sim_length)} \]

for \( t \) in range(sim_length-1):
    \[ X, M = \text{econ}.\text{gen_aggregates()} \]
    \[ X_{\text{vec}[t]} = X \]
    \[ M_{\text{vec}[t]} = M \]

    \[ \text{econ}.\text{update_beliefs}(X, M) \]
    \[ \text{econ}.\text{update}_\theta(w_{\text{shocks}[t]}) \]

    \[ \mu_{\text{vec}[t+1]} = \text{econ.} \mu \]
    \[ \gamma_{\text{vec}[t+1]} = \text{econ.} \gamma \]
    \[ \theta_{\text{vec}[t+1]} = \text{econ.} \theta \]

# Record final values of aggregates
\[ X, M = \text{econ}.\text{gen_aggregates()} \]
\[ X_{\text{vec}[-1]} = X \]
\[ M_{\text{vec}[-1]} = M \]

First lets see how well \( \mu \) tracks \( \theta \) in these simulations

```python
fig, ax = plt.subplots(figsize=(9, 6))
ax.plot(range(sim_length), \theta_{\text{vec}}, alpha=0.6, lw=2, label=r"$\theta$")
ax.plot(range(sim_length), \mu_{\text{vec}}, alpha=0.6, lw=2, label=r"$\mu$")
ax.legend(fontsize=16)
ax.grid()
plt.show()
```
QuantEcon.lectures-python3 PDF, Release 2018-Aug-8

Now lets plot the whole thing together
fig, axes = plt.subplots(4, 1, figsize=(12, 20))
# Add some spacing
fig.subplots_adjust(hspace=0.3)
series = (θ_vec, µ_vec, γ_vec, M_vec)
names = r'$\theta$', r'$\mu$', r'$\gamma$', r'$M$'
for ax, vals, name in zip(axes, series, names):
# determine suitable y limits
s_max, s_min = max(vals), min(vals)
s_range = s_max - s_min
y_max = s_max + s_range * 0.1
y_min = s_min - s_range * 0.1
ax.set_ylim(y_min, y_max)
# Plot series
ax.plot(range(sim_length), vals, alpha=0.6, lw=2)
ax.set_title(f"time series for {name}", fontsize=16)
ax.grid()
plt.show()

928

Chapter 7. Multiple Agent Models


7.9. Uncertainty Traps
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)

7.10 The Aiyagari Model

7.10.1 Overview

In this lecture we describe the structure of a class of models that build on work by Truman Bewley [Bew77]
We begin by discussing an example of a Bewley model due to Rao Aiyagari
The model features
• Heterogeneous agents
• A single exogenous vehicle for borrowing and lending
• Limits on amounts individual agents may borrow

The Aiyagari model has been used to investigate many topics, including
• precautionary savings and the effect of liquidity constraints [Aiy94]
• risk sharing and asset pricing [HL96]
• the shape of the wealth distribution [BBZ15]
• etc., etc., etc.

References

The primary reference for this lecture is [Aiy94]
A textbook treatment is available in chapter 18 of [LS18]
A continuous time version of the model by SeHyoun Ahn and Benjamin Moll can be found here

7.10.2 The Economy

Households

Infinitely lived households / consumers face idiosyncratic income shocks
A unit interval of ex ante identical households face a common borrowing constraint
The savings problem faced by a typical household is

\[
\max \mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)
\]
subject to
\[ a_{t+1} + c_t \leq wz_t + (1 + r)a_t \quad c_t \geq 0, \quad \text{and} \quad a_t \geq -B \]

where
- \( c_t \) is current consumption
- \( a_t \) is assets
- \( z_t \) is an exogenous component of labor income capturing stochastic unemployment risk, etc.
- \( w \) is a wage rate
- \( r \) is a net interest rate
- \( B \) is the maximum amount that the agent is allowed to borrow

The exogenous process \( \{z_t\} \) follows a finite state Markov chain with given stochastic matrix \( P \)

The wage and interest rate are fixed over time

In this simple version of the model, households supply labor inelastically because they do not value leisure

### 7.10.3 Firms

Firms produce output by hiring capital and labor

Firms act competitively and face constant returns to scale

Since returns to scale are constant the number of firms does not matter

Hence we can consider a single (but nonetheless competitive) representative firm

The firms output is
\[ Y_t = AK_t^\alpha N^{1-\alpha} \]

where
- \( A \) and \( \alpha \) 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 firms problem is
\[ \max_{K,N} \left\{ AK_t^\alpha N^{1-\alpha} - (r + \delta)K - wN \right\} \]

The parameter \( \delta \) is the depreciation rate

From the first-order condition with respect to capital, the firms inverse demand for capital is
\[ r = A\alpha \left( \frac{N}{K} \right)^{1-\alpha} - \delta \] (7.86)
Using this expression and the firms first-order condition for labor, we can pin down the equilibrium wage rate as a function of \( r \) as

\[
w(r) = A(1 - \alpha)(A\alpha/(r + \delta))^{\alpha/(1 - \alpha)}
\]  

(7.87)

**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 (7.86) and a wage rate \( w(r) \) as given in (7.87)
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

**7.10.4 Code**

Let’s look at how we might compute such an equilibrium in practice

To solve the households dynamic programming problem well 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 will adopt unless otherwise specified.

```python
import numpy as np
from numba import jit

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 $S = \{0, \ldots, n\}$. To this end, $(a_i, z_i)$ index pairs are mapped to $s_i$ indices according to the rule

    $s_i = a_i \times z_{\text{size}} + z_i$

    To invert this map, use

    $a_i = s_i \div z_{\text{size}} \quad \text{(integer division)}$
    $z_i = s_i \mod z_{\text{size}}$

    ""

    def __init__(self,
                 r=0.01,  # interest rate
                 w=1.0,   # wages
                 $\beta=0.96$,  # discount factor
                 a_min=1e-10,
                 $\Pi=[[0.9, 0.1], [0.1, 0.9]]$,  # 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.$\beta$ = r, w, $\beta$
        self.a_min, self.a_max, self.a_size = a_min, a_max, a_size

        self.$\Pi$ = np.asarray($\Pi$)
        self.z_vals = np.asarray(z_vals)
        self.z_size = len(z_vals)
```

7.10. The Aiyagari Model
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.PI)

    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, PI):
    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] = PI[z_i, next_z_i]
As a first example of what we can do, let's compute and plot an optimal accumulation policy at fixed prices:

```python
@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
```

```python
import quantecon as qe
import matplotlib.pyplot as plt
from quantecon.markov import DiscreteDP

# 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.B)

# 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()
```

Here's the output: 7.10. The Aiyagari Model
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

\[
\begin{align*}
A &= 1.0 \\
N &= 1.0 \\
\alpha &= 0.33 \\
\beta &= 0.96 \\
\delta &= 0.05
\end{align*}
\]
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

```python
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()
```

Here's the corresponding plot

---

### 7.11 Default Risk and Income Fluctuations

**Contents**

- Default Risk and Income Fluctuations
7.11.1 Overview

This lecture computes versions of Arellanos [Are08] model of sovereign default

The model describes interactions among default risk, output, and an equilibrium interest rate that includes a premium for endogenous default risk

The decision maker is a government of a small open economy that borrows from risk-neutral foreign creditors

The foreign lenders must be compensated for default risk

The government borrows and lends abroad in order to smooth the consumption of its citizens

The government repays its debt only if it wants to, but declining to pay has adverse consequences

The interest rate on government debt adjusts in response to the state-dependent default probability chosen by government

The model yields outcomes that help interpret sovereign default experiences, including

- countercyclical interest rates on sovereign debt
- countercyclical trade balances
- high volatility of consumption relative to output

Notably, long recessions caused by bad draws in the income process increase the government's incentive to default

This can lead to

- spikes in interest rates
- temporary losses of access to international credit markets
- large drops in output, consumption, and welfare
- large capital outflows during recessions

Such dynamics are consistent with experiences of many countries
7.11.2 Structure

In this section we describe the main features of the model.

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

\[
E \sum_{t=0}^{\infty} \beta^t u(c_t)
\]  
(7.88)

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 (7.88).

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.

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

\[ c = y + B - q(B', y)B' \] (7.89)

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.

**Financial Markets**

Foreign creditors

- are risk neutral
- know the domestic output stochastic process \( \{ y_t \} \) and observe \( y_t, y_{t-1}, \ldots \), at time \( t \)
- can borrow or lend without limit in an international credit market at a constant international interest rate \( r \)
- receive full payment if the government chooses to pay
- receive zero if the government defaults on its one-period debt due

When a government is expected to default next period with probability \( \delta \), the expected value of a promise to pay one unit of consumption next period is \( 1 - \delta \).

Therefore, the discounted expected value of a promise to pay \( B \) next period is

\[ q = \frac{1 - \delta}{1 + r} \] (7.90)

Next we turn to how the government in effect chooses the default probability \( \delta \).

**Governments decisions**

At each point in time \( t \), the government chooses between

1. defaulting
2. meeting its current obligations and purchasing or selling an optimal quantity of one-period sovereign debt

Defaulting means declining to repay all of its current obligations.

If the government defaults in the current period, then consumption equals current output.

But a sovereign default has two consequences:

1. Output immediately falls from \( y \) to \( h(y) \), where \( 0 \leq h(y) \leq y \)
• it returns to $y$ only after the country regains access to international credit markets

2. The country loses access to foreign credit markets

**Reentering international credit market**

While in a state of default, the economy regains access to foreign credit in each subsequent period with probability $\theta$

**7.11.3 Equilibrium**

Informally, an equilibrium is a sequence of interest rates on its sovereign debt, a stochastic sequence of government default decisions and an implied flow of household consumption such that

1. Consumption and assets satisfy the national budget constraint

2. The government maximizes household utility taking into account
   - the resource constraint
   - the effect of its choices on the price of bonds
   - consequences of defaulting now for future net output and future borrowing and lending opportunities

3. The interest rate on the government's debt includes a risk-premium sufficient to make foreign creditors expect on average to earn the constant risk-free international interest rate

To express these ideas more precisely, consider first the choices of the government, which

1. enters a period with initial assets $B$, or what is the same thing, initial debt to be repaid now of $-B$

2. observes current output $y$, and

3. chooses either
   (a) to default, or
   (b) to pay $-B$ and set next periods 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

\[ \delta(B', y) := \int \mathbb{1}\{v_c(B', y') < v_d(y')\} p(y, y') dy' \] \hspace{1cm} (7.91)

Given zero profits for foreign creditors in equilibrium, we can combine (7.90) and (7.91) to pin down the bond price function:

\[ q(B', y) = \frac{1 - \delta(B', y)}{1 + r} \] \hspace{1cm} (7.92)

**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 (7.92)
7.11.4 Computation

Lets now compute an equilibrium of Arellanos model

The equilibrium objects are the value function $v(B, y)$, the associated default decision rule, and the pricing function $q(B', y)$

Well use our code to replicate Arellanos results

After that well perform some additional simulations

The majority of the code below was written by Chase Coleman

It uses a slightly modified version of the algorithm recommended by Arellano

- The appendix to [Are08] recommends value function iteration until convergence, updating the price, and then repeating
- Instead, we update the bond price at every value function iteration step

The second approach is faster and the two different procedures deliver very similar results

Here is a more detailed description of our algorithm:

1. Guess a value function $v(B, y)$ and price function $q(B', y)$
2. At each pair $(B, y)$,
   - update the value of defaulting $v_d(y)$
   - update the value of continuing $v_c(B, y)$
3. Update the value function $v(B, y)$, the default rule, the implied ex ante default probability, and the price function
4. Check for convergence. If converged, stop. If not, go to step 2.

We use simple discretization on a grid of asset holdings and income levels

The output process is discretized using Tauchens quadrature method

Numba has been used in two places to speed up the code

The code can be found in the file arellano_vfi.py but we repeat it here for convenience

(Results and discussion follow the code)
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
-----------
\( \beta \) : float
    Time discounting parameter
\( \gamma \) : float
    Risk-aversion parameter
\( r \) : float
    Int lending rate
\( \rho \) : float
    Persistence in the income process
\( \eta \) : float
    Standard deviation of the income process
\( \theta \) : float
    Probability of re-entering financial markets in each period
\( n_y \) : int
    Number of points in y grid
\( n_B \) : int
    Number of points in B grid
tol : float
    Error tolerance in iteration
maxit : int
    Maximum number of iterations

```python
def __init__(self,
    \( \beta = .953 \),            # time discount rate
    \( \gamma = 2. \),              # risk aversion
    \( r = .017 \),                # international interest rate
    \( \rho = .945 \),              # persistence in output
    \( \eta = .025 \),              # st dev of output shock
    \( \theta = .282 \),            # prob of regaining access
    \( n_y = 21 \),                # number of points in y grid
    \( n_B = 251 \),               # number of points in B grid
    tol=1e-8,                     # error tolerance in iteration
    maxit=10000):                 # Maximum number of iterations

    # Save parameters
    self.\( \beta \), self.\( \gamma \), self.\( r \) = \( \beta \), \( \gamma \), \( r \)
    self.\( \rho \), self.\( \eta \), self.\( \theta \) = \( \rho \), \( \eta \), \( \theta \)
    self.\( n_y \), self.\( n_B \) = \( n_y \), \( n_B \)

    # Create grids and discretize Markov process
    self.Bgrid = np.linspace(-.45, .45, nB)
    self.mc = qe.markov.tauchen(\( \rho \), \( \eta \), 3, ny)
    self.ygrid = np.exp(self.mc.state_values)
    self.Py = self.mc.P
```

7.11. Default Risk and Income Fluctuations
# Output when in default
ymean = np.mean(self.ygrid)
self.def_y = np.minimum(0.969 * ymean, self.ygrid)

# Allocate memory
self.Vd = np.zeros(ny)
self.Vc = np.zeros((ny, nB))
self.V = np.zeros((ny, nB))
self.Q = np.ones((ny, nB)) * .95  # Initial guess for prices
self.default_prob = np.empty((ny, nB))

# Compute the value functions, prices, and default prob
self.solve(tol=tol, maxit=maxit)
# Compute the optimal savings policy conditional on no default
self.compute_savings_policy()

def solve(self, tol=1e-8, maxit=10000):
    # Iteration Stuff
    it = 0
    dist = 10.

    # Alloc memory to store next iterate of value function
    V_upd = np.zeros((self.ny, self.nB))

    # == Main loop == #
    while dist > tol and maxit > it:
        # Compute expectations for this iteration
        Vs = self.V, self.Vd, self.Vc
        EV, EVd, EVc = (self.Py @ v for v in Vs)

        # Run inner loop to update value functions Vc and Vd.
        # Note that Vc and Vd are updated in place. Other objects
        # are not modified.
        _inner_loop(self.ygrid, self.def_y,
                    self.Bgrid, self.Vd, self.Vc,
                    EVc, EVd, EV, self.Q,
                    self.β, self.θ, self.γ)

        # Update prices
        Vd_compat = np.repeat(self.Vd, self.nB).reshape(self.ny, self.nB)
default_states = Vd_compat > self.Vc
self.default_prob[:, :] = self.Py @ default_states
self.Q[:, :] = (1 - self.default_prob)/(1 + self.r)

        # Update main value function and distance
        V_upd[:, :] = np.maximum(self.Vc, Vd_compat)
dist = np.max(np.abs(V_upd - self.V))
sself.V[:, :] = V_upd[:, :]

        it += 1
        if it % 25 == 0:
            print(f"Running iteration {it} with dist of {dist}"
return None

def compute_savings_policy(self):
    
    Compute optimal savings B' conditional on not defaulting. The policy is recorded as an index value in Bgrid.
    
    # Allocate memory
    self.next_B_index = np.empty((self.ny, self.nB))
    EV = self.Py @ self.V

    _compute_savings_policy(self.ygrid, self.Bgrid, self.Q, EV, self.γ, self.β, self.next_B_index)

def simulate(self, T, y_init=None, B_init=None):
    
    Simulate time series for output, consumption, B'.
    
    # Find index i such that Bgrid[i] is near 0
    zero_B_index = np.searchsorted(self.Bgrid, 0)

    if y_init is None:
        # Set to index near the mean of the ygrid
        y_init = np.searchsorted(self.ygrid, self.ygrid.mean())
    if B_init is None:
        B_init = zero_B_index

    in_default = False

    y_sim_indices = self.mc.simulate_indices(T, init=y_init)
    B_sim_indices = np.empty(T, dtype=np.int64)
    B_sim_indices[0] = B_init

    q_sim = np.empty(T)
    in_default_series = np.zeros(T, dtype=np.int64)

    for t in range(T-1):
        yi, Bi = y_sim_indices[t], B_sim_indices[t]
        if not in_default:
            if self.Vc[yi, Bi] < self.Vd[yi]:
                in_default = True
                Bi_next = zero_B_index
            else:
                new_index = self.next_B_index[yi, Bi]
                Bi_next = new_index
        else:
            in_default_series[t] = 1
            Bi_next = zero_B_index
            if random.uniform(0, 1) < self.θ:
                in_default = False
                B_sim_indices[t+1] = Bi_next
                q_sim[t] = self.Q[yi, int(Bi_next)]
q_sim[-1] = q_sim[-2]  # Extrapolate for the last price
return_vecs = (self.ygrid[y_sim_indices],
                self.Bgrid[B_sim_indices],
                q_sim,
                in_default_series)

        return return_vecs

@jit(nopython=True)
def u(c, γ):
    return c**(1-γ)/(1-γ)

@jit(nopython=True)
def _inner_loop(ygrid, def_y, Bgrid, Vd, Vc, EVc, EVd, EV, qq, β, θ, γ):
    """
    This is a numba version of the inner loop of the solve in the
    Arellano class. It updates Vd and Vc in place.
    """
    ny, nB = len(ygrid), len(Bgrid)
    zero_ind = nB // 2  # Integer division
    for iy in range(ny):
        y = ygrid[iy]  # Pull out current y
        # Compute Vd
        Vd[iy] = u(def_y[iy], γ) + β * (θ * EVc[iy, zero_ind] + (1 - θ) * EVd[iy])
        # Compute Vc
        for ib in range(nB):
            B = Bgrid[ib]  # Pull out current B
            current_max = -1e14
            for ib_next in range(nB):
                c = max(y - qq[iy, ib_next] * Bgrid[ib_next] + B, 1e-14)
                m = u(c, γ) + β * EV[iy, ib_next]
                if m > current_max:
                    current_max = m
            Vc[iy, ib] = current_max

    return None

@jit(nopython=True)
def _compute_savings_policy(ygrid, Bgrid, Q, EV, γ, β, next_B_index):
    # Compute best index in Bgrid given iy, ib
    ny, nB = len(ygrid), len(Bgrid)
    for iy in range(ny):
        y = ygrid[iy]
        for ib in range(nB):
B = Bgrid[ib]
current_max = -1e10
for ib_next in range(nB):
    c = max(y - Q[iy, ib_next] * Bgrid[ib_next] + B, 1e-14)
    m = u(c, \gamma) + \beta * EV[iy, ib_next]
    if m > current_max:
        current_max = m
        current_max_index = ib_next
    next_B_index[iy, ib] = current_max_index

return None

### 7.11.5 Results

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

In what follows, all results are computed using Arellanos 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 \((n_y, n_B = 21, 251)\) in order to match Arrelanos findings.

Here’s the same relationships computed on a finer grid \((n_y, n_B = 51, 551)\)
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 [Are08]
We can use the results of the computation to study the default probability $\delta(B', y)$ defined in (7.91)

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.
7.11.6 Exercises

Exercise 1

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

- Use the parameter values listed as defaults in the \texttt{__init__} method of the \texttt{Arellano\_Economy}
- The time series will of course vary depending on the shock draws

7.11.7 Solutions

Compute the value function, policy and equilibrium prices

```python
import matplotlib.pyplot as plt

ae = Arellano_Economy(\beta=.953,  # time discount rate
                       \gamma=2.,     # risk aversion
                       r=0.017,      # international interest rate
                       \rho=.945,     # persistence in output
                       \eta=0.025,    # st dev of output shock
                       \theta=0.282,  # prob of regaining access
                       ny=21,        # number of points in y grid
                       nB=251,       # number of points in B grid
                       tol=1e-8,     # error tolerance in iteration
                       maxit=10000)
```

Running iteration 25 with dist of 0.3432423298900247
Running iteration 50 with dist of 0.0983915579847886
Running iteration 75 with dist of 0.0292120959165606
Running iteration 100 with dist of 0.00874510696905162
Running iteration 125 with dist of 0.002623141215583047
Running iteration 150 with dist of 0.0007871926699110077
Running iteration 175 with dist of 0.00023625911163094315
Running iteration 200 with dist of 7.091000629344535e-05
Running iteration 225 with dist of 2.12828211445526e-05
Running iteration 250 with dist of 6.387802962137812e-06
Running iteration 275 with dist of 1.9172289675850607e-06
Running iteration 300 with dist of 5.754352905285032e-07
Running iteration 325 with dist of 1.7271061736323645e-07
Running iteration 350 with dist of 5.187215409022974e-08
Running iteration 375 with dist of 1.55538125000264e-08

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

```python
# Create "Y High" and "Y Low" values as 5% devs from mean
high, low = np.mean(ae.ygrid) + 1.05, np.mean(ae.ygrid) + 0.95
iy_high, iy_low = (np.searchsorted(ae.ygrid, x) for x in (high, low))

fig, ax = plt.subplots(figsize=(10, 6.5))
ax.set_title("Bond price schedule $q(y, B')$")
```

7.11. Default Risk and Income Fluctuations
# Extract a suitable plot grid

```python
x = []
q_low = []
q_high = []
for i in range(ae.nB):
    b = ae.Bgrid[i]
    if -0.35 <= b <= 0:  # To match fig 3 of Arellano
        x.append(b)
        q_low.append(ae.Q[iy_low, i])
        q_high.append(ae.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("$B'$")
```

```
plt.show()
```

Draw a plot of the value functions

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

fig, ax = plt.subplots(figsize=(10, 6.5))
ax.set_title("Value Functions")
```
Draw a heat map for default probability

```python
xx, yy = ae.Bgrid, ae.ygrid
zz = ae.default_prob

# Create figure
fig, ax = plt.subplots(figsize=(10, 6.5))
hm = ax.pcolormesh(xx, yy, zz)
cax = fig.add_axes([.92, .i, .02, .8])
fig.colorbar(hm, cax=cax)
ax.axis([xx.min(), 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.

```python
T = 250
y_vec, B_vec, q_vec, default_vec = ae.simulate(T)

# 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()
7.12 Globalization and Cycles
This lecture is coauthored with Chase Coleman

### 7.12.1 Overview

In this lecture, we review the paper *Globalization and Synchronization of Innovation Cycles* by Kiminori Matsuyama, Laura Gardini and Iryna Sushko

This model helps us understand several interesting stylized facts about the world economy.

One of these is synchronized business cycles across different countries.

Most existing models that generate synchronized business cycles do so by assumption, since they tie output in each country to a common shock.

They also fail to explain certain features of the data, such as the fact that the degree of synchronization tends to increase with trade ties.

By contrast, in the model we consider in this lecture, synchronization is both endogenous and increasing with the extent of trade integration.

In particular, as trade costs fall and international competition increases, innovation incentives become aligned and countries synchronize their innovation cycles.

#### Background

The model builds on work by Judd [*Jud85*], Deneckner and Judd [*DJ92*] and Helpman and Krugman [*HK85*] by developing a two country model with trade and innovation.

On the technical side, the paper introduces the concept of coupled oscillators to economic modeling.

As we will see, coupled oscillators arise endogenously within the model.

Below we review the model and replicate some of the results on synchronization of innovation across countries.
7.12.2 Key Ideas

It is helpful to begin with an overview of the mechanism

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

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 others markets

As illustrated below, this leads to synchonization of business cycles across the two countries

7.12.3 Model

Let’s write down the model more formally

(The treatment is relatively terse since full details can be found in the original paper)

Time is discrete with \( t = 0, 1, \ldots \)

There are two countries indexed by \( j \) or \( k \)

In each country, a representative household inelastically supplies \( L_j \) units of labor at wage rate \( w_{jt} \)

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

\[ X^{1\frac{1}{\sigma}}_{k,t} = \int_{\Omega_t} [x_{k,t}(\nu)]^{\frac{1}{\sigma}} d\nu \]

Here \( x_{k,t}(\nu) \) is the total amount of a differentiated good \( \nu \in \Omega_t \) that is produced. The parameter \( \sigma > 1 \) is the direct partial elasticity of substitution between a pair of varieties and \( \Omega_t \) is the set of varieties available in period \( t \).

We can split the varieties into those which are supplied competitively and those supplied monopolistically; that is, \( \Omega_t = \Omega^c_t + \Omega^m_t \).

**Prices**

Demand for differentiated inputs is

\[ x_{k,t}(\nu) = \left( \frac{p_{k,t}(\nu)}{P_{k,t}} \right)^{-\sigma} \frac{\alpha L_k}{P_{k,t}} \]

Here

- \( p_{k,t}(\nu) \) is the price of the variety \( \nu \) and
- \( P_{k,t} \) is the price index for differentiated inputs in \( k \), defined by

\[ [P_{k,t}]^{1-\sigma} = \int_{\Omega_t} [p_{k,t}(\nu)]^{1-\sigma} d\nu \]

The price of a variety also depends on the origin, \( j \), and destination, \( k \), of the goods because shipping varieties between countries incurs an iceberg trade cost \( \tau_{j,k} \). Thus the effective price in country \( k \) of a variety \( \nu \) produced in country \( j \) becomes \( p_{k,t}(\nu) = \tau_{j,k} p_{j,t}(\nu) \).

Using these expressions, we can derive the total demand for each variety, which is

\[ D_{j,t}(\nu) = \sum_k \tau_{j,k} x_{k,t}(\nu) = \alpha A_{j,t}(p_{j,t}(\nu))^{-\sigma} \]

where

\[ A_{j,t} := \sum_k \frac{\rho_{j,k} L_k}{(P_{k,t})^{1-\sigma}} \quad \text{and} \quad \rho_{j,k} = (\tau_{j,k})^{1-\sigma} \leq 1 \]
It is assumed that $\tau_{1,1} = \tau_{2,2} = 1$ and $\tau_{1,2} = \tau_{2,1} = \tau$ for some $\tau > 1$, so that
\[ \rho_{1,2} = \rho_{2,1} = \rho := \tau^{1-\sigma} < 1 \]

The value $\rho \in [0, 1)$ is a proxy for the degree of globalization.

Producing one unit of each differentiated variety requires $\psi$ units of labor, so the marginal cost is equal to $\psi$ for $\nu \in \Omega_{j,t}$.

Additionally, all competitive varieties will have the same price (because of equal marginal cost), which means that, for all $\nu \in \Omega^c$,
\[ p_{j,t}(\nu) = p_{j,t}^c := \psi \quad \text{and} \quad D_{j,t} = y_{j,t}^c := \alpha A_{j,t}(p_{j,t}^c)^{-\sigma} \]

Monopolists will have the same marked-up price, so, for all $\nu \in \Omega^m$,
\[ p_{j,t}(\nu) = p_{j,t}^m := \frac{\psi}{1 - \frac{1}{\sigma}} \quad \text{and} \quad D_{j,t} = y_{j,t}^m := \alpha A_{j,t}(p_{j,t}^m)^{-\sigma} \]

Define
\[ \theta := \frac{p_{j,t}^c y_{j,t}^c}{p_{j,t}^m y_{j,t}^m} = \left(1 - \frac{1}{\sigma}\right)^{1-\sigma} \]

Using the preceding definitions and some algebra, the price indices can now be rewritten as
\[ \left(\frac{P_{k,t}}{\psi}\right)^{1-\sigma} = M_{k,t} + \rho M_{j,t} \quad \text{where} \quad M_{j,t} := N_{j,t}^c + \frac{N_{j,t}^m}{\theta} \]

The symbols $N_{j,t}^c$ and $N_{j,t}^m$ will denote the measures of $\Omega^c$ and $\Omega^m$ respectively.

**New Varieties**

To introduce a new variety, a firm must hire $f$ units of labor per variety in each country.

Monopolist profits must be less than or equal to zero in expectation, so
\[ N_{j,t}^m \geq 0, \quad \pi_{j,t}^m := (p_{j,t}^m - \psi)y_{j,t}^m - f \leq 0 \quad \text{and} \quad \pi_{j,t}^m N_{j,t}^m = 0 \]

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

**Law of Motion**

With $\delta$ as the exogenous probability of a variety becoming obsolete, the dynamic equation for the measure of firms becomes
\[ N_{j,t+1}^c = \delta(N_{j,t}^c + N_{j,t}^m) = \delta(N_{j,t}^c + \theta(M_{j,t} - N_{j,t}^c)) \]
7.12. Globalization and Cycles 965

As we will see, this depends on initial conditions. We will focus in particular on whether or not innovation cycles synchronize across the two countries.

Let's try simulating some of these trajectories.

The normalized measure of varieties is

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

We also use $s_j := \frac{L_j}{L_1 + L_2}$ to be the share of labor employed in country $j$.

We can use these definitions and the preceding expressions to obtain a law of motion for $n_t := (n_{1,t}, n_{2,t})$.

In particular, given an initial condition, $n_0 = (n_{1,0}, n_{2,0}) \in \mathbb{R}_+^2$, the equilibrium trajectory, $\{n_t\}_{t=0}^\infty = \{(n_{1,t}, n_{2,t})\}_{t=0}^\infty$, is obtained by iterating on $n_{t+1} = F(n_t)$ where $F : \mathbb{R}_+^2 \to \mathbb{R}_+^2$ is given by

$$ F(n_t) = \begin{cases} 
  \left( \delta(\theta s_1(\rho) + (1 - \rho)n_{1,t}), \delta(\theta s_2(\rho) + (1 - \rho)n_{2,t}) \right) & \text{for } n_t \in D_{LL} \\
  \left( \delta n_{1,t}, \delta n_{2,t} \right) & \text{for } n_t \in D_{HH} \\
  \left( \delta(\theta h_2(n_{1,t}) + (1 - \theta)n_{2,t}), \delta(n_{1,t}, (1 - \theta)n_{2,t}) \right) & \text{for } n_t \in D_{HL} \\
  \left( \delta h_1(n_{2,t}) + (1 - \theta)n_{1,t}, \delta n_{2,t} \right) & \text{for } n_t \in D_{LH} 
\end{cases} $$

Here

- $D_{LL} := \{(n_1, n_2) \in \mathbb{R}_+^2 | n_j \leq s_j(\rho)\}$
- $D_{HH} := \{(n_1, n_2) \in \mathbb{R}_+^2 | n_j \geq h_j(\rho)\}$
- $D_{HL} := \{(n_1, n_2) \in \mathbb{R}_+^2 | n_1 \geq s_1(\rho) \text{ and } n_2 \leq h_2(n_1)\}$
- $D_{LH} := \{(n_1, n_2) \in \mathbb{R}_+^2 | n_1 \leq h_1(n_2) \text{ and } n_2 \geq s_2(\rho)\}$

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

$$ h_j(n_k)^2 + \left( \rho + \frac{1}{\rho} \right)n_k - s_j - s_k \right) h_j(n_k) + \left( n_k^2 - \frac{s_j n_k}{\rho} - s_k n_k \rho \right) = 0 $$

### 7.12.4 Simulation

Let's try simulating some of these trajectories.

We will focus in particular on whether or not innovation cycles synchronize across the two countries.

As we will see, this depends on initial conditions.

For some parameterizations, synchronization will occur for most initial conditions, while for others synchronization will be rare.
The computational burden of testing synchronization across many initial conditions is not trivial.

In order to make our code fast, we will use just in time compiled functions that will get called and handled by our class.

These are the @jit statements that you see below (review *this lecture* if you don’t recall how to use JIT compilation).

Here’s the main body of code:

```python
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

from numba import jit, vectorize

@jit(nopython=True)
def _hj(j, nk, s1, s2, \theta, \delta, \rho):
    
    """
    If we expand the implicit function for h_j(n_k) then we find that
    it is a 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 = ((\rho + 1 / \rho) * nk - sj - sk)
    c = (nk * nk - (sj + nk) / \rho - sk * \rho * 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_\rho, s2_\rho, s1, s2, \theta, \delta, \rho):
    """Determine whether (n1, n2) is in the set DLL"
    return (n1 <= s1_\rho) and (n2 <= s2_\rho)

@jit(nopython=True)
def DHH(n1, n2, s1_\rho, s2_\rho, s1, s2, \theta, \delta, \rho):
    """Determine whether (n1, n2) is in the set DHH"
    return (n1 >= _hj(1, n2, s1, s2, \theta, \delta, \rho)) and (n2 >= _hj(2, n1, s1, s2, \theta, \delta, \rho))

@jit(nopython=True)
```

Chapter 7. Multiple Agent Models
def DHL(n1, n2, s1_, s2_, s1, s2, \(\theta, \delta, \rho\)):
    """Determine whether \((n_1, n_2)\) is in the set DHL"
    return (n1 >= s1_) and (n2 <= _hj(2, n1, s1, s2, \(\theta, \delta, \rho\)))

@jit(nopython=True)
def DLH(n1, n2, s1_, s2_, s1, s2, \(\theta, \delta, \rho\)):
    """Determine whether \((n_1, n_2)\) is in the set DLH"
    return (n1 <= _hj(1, n2, s1, s2, \(\theta, \delta, \rho\))) and (n2 >= s2_)

@jit(nopython=True)
def one_step(n1, n2, s1_, s2_, s1, s2, \(\theta, \delta, \rho\)):
    """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.
    """
    if DLL(n1, n2, s1_, s2_, s1, s2, \(\theta, \delta, \rho\)):
        n1_tp1 = \(\delta \ast (\theta \ast s1_ + (1 - \theta) \ast n1)\)
        n2_tp1 = \(\delta \ast (\theta \ast s2_ + (1 - \theta) \ast n2)\)
    elif DHH(n1, n2, s1_, s2_, s1, s2, \(\theta, \delta, \rho\)):
        n1_tp1 = \(\delta \ast n1\)
        n2_tp1 = \(\delta \ast n2\)
    elif DHL(n1, n2, s1_, s2_, s1, s2, \(\theta, \delta, \rho\)):
        n1_tp1 = \(\delta \ast n1\)
        n2_tp1 = \(\delta \ast _hj(2, n1, s1, s2, \theta, \delta, \rho) + (1 - \theta) \ast n2)\)
    elif DLH(n1, n2, s1_, s2_, s1, s2, \(\theta, \delta, \rho\)):
        n1_tp1 = \(\delta \ast _hj(1, n2, s1, s2, \theta, \delta, \rho) + (1 - \theta) \ast n1)\)
        n2_tp1 = \(\delta \ast n2\)
    return n1_tp1, n2_tp1

@jit(nopython=True)
def n_generator(n1_0, n2_0, s1_, s2_, s1, s2, \(\theta, \delta, \rho\)):
    """Given an initial condition, continues to yield new values of
    \(n_1\) and \(n_2\)"
    n1_t, n2_t = n1_0, n2_0
    while True:
        n1_tp1, n2_tp1 = one_step(n1_t, n2_t, s1_, s2_, s1, s2, \(\theta, \delta, \rho\))
        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_, s2_, s1, s2, \(\theta, \delta, \rho\), 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 by 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 they 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 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
    """
    # 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_t1, n2_t1 = 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 they two economies end up synchronized
    pers_2_sync : scalar(Int)
        The number of periods required until they synchronized
    ""
    # Unpack parameters
    s1, s2, theta, delta, rho = self._unpack_params()
    s1_rho, s2_rho = self.s1_rho, self.s2_rho

    return _pers_till_sync(n1_0, n2_0, s1_rho, s2_rho, s1, s2, theta, delta, rho, 
                            maxiter, npers)

    def create_attraction_basis(self, maxiter=250, npers=3, npts=50):
        ""
        Creates an attraction basis for values of \( n \) on \([0, 1] \times [0, 1] \) with
        \( npts \) in each dimension
        ""
        # Unpack parameters
        s1, s2, theta, delta, rho = self._unpack_params()
        s1_rho, s2_rho = self.s1_rho, self.s2_rho
ab = _create_attraction_basis(s1_ρ, s2_ρ, s1, s2, θ, δ,
ρ, maxiter, npers, npts)

return ab

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

Heres the function

    def plot_timeseries(n1_0, n2_0, s1=0.5, s2=2.5, θ=0.7, δ=0.2, ax=None, title='Not Synchronized'):
        Plot a single time series with initial conditions
# 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()

Lets see what we get
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.

**Basin of Attraction**

Next let's study the initial conditions that lead to synchronized cycles more systematically.

We generate time series from a large collection of different initial conditions and mark those conditions with different colors according to whether synchronization occurs or not.

The next display shows exactly this for four different parameterizations (one for each subfigure).

Dark colors indicate synchronization, while light colors indicate failure to synchronize.
As you can see, larger values of $\rho$ translate to more synchronization.

You are asked to replicate this figure in the exercises.

In the solution to the exercises, you'll also find a figure with sliders, allowing you to experiment with different parameters.

Here's one snapshot from the interactive figure.
7.12.5 Exercises

Exercise 1

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

7.12.6 Solutions

These solutions are written by Chase Coleman

```python
import matplotlib.pyplot as plt

def plot_attraction_basis(s1=0.5, \theta=2.5, \delta=0.7, \rho=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, \theta, \delta, \rho)
    ab = model.create_attraction_basis(npts=npts)
```

7.12. Globalization and Cycles
```
cf = ax.pcolormesh(unitrange, unitrange, ab, cmap="viridis")

    return ab, cf

fig = plt.figure(figsize=(14, 12))

# Left - Bottom - Width - Height
ax0 = fig.add_axes((0.05, 0.475, 0.38, 0.35), label="axes0")
ax1 = fig.add_axes((0.5, 0.475, 0.38, 0.35), label="axes1")
ax2 = fig.add_axes((0.05, 0.05, 0.38, 0.35), label="axes2")
ax3 = fig.add_axes((0.5, 0.05, 0.38, 0.35), label="axes3")

params = [[0.5, 2.5, 0.7, 0.2],
           [0.5, 2.5, 0.7, 0.4],
           [0.5, 2.5, 0.7, 0.6],
           [0.5, 2.5, 0.7, 0.8]]

ab0, cf0 = plot_attraction_basis(*params[0], npts=500, ax=ax0)
ab1, cf1 = plot_attraction_basis(*params[1], npts=500, ax=ax1)
ab2, cf2 = plot_attraction_basis(*params[2], npts=500, ax=ax2)
ab3, cf3 = plot_attraction_basis(*params[3], npts=500, ax=ax3)

cbar_ax = fig.add_axes([0.9, 0.075, 0.03, 0.725])
plt.colorbar(cf0, cax=cbar_ax)

ax0.set_title(r"$s_1=0.5$, \theta=2.5$, $\delta=0.7$, $\rho=0.2$",
               fontsize=22)
ax1.set_title(r"$s_1=0.5$, $\theta=2.5$, $\Delta=0.7$, $\rho=0.4$",
               fontsize=22)
ax2.set_title(r"$s_1=0.5$, $\theta=2.5$, $\Delta=0.7$, $\rho=0.6$",
               fontsize=22)
ax3.set_title(r"$s_1=0.5$, $\theta=2.5$, $\Delta=0.7$, $\rho=0.8$",
               fontsize=22)

fig.suptitle("Synchronized versus Asynchronized 2-cycles",
             x=0.475, y=0.915, size=26)
plt.show()
```
Additionally, instead of just seeing 4 plots at once, we might want to manually be able to change $\rho$ and see how it affects the plot in real time. Below we use an interactive plot to do this.

Note, interactive plotting requires the ipywidgets module to be installed and enabled.

```python
from ipywidgets import interact
def interact_attraction_basis(\rho=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, \theta, \delta = 0.5, 2.5, 0.75
    model = MSGSync(s1, \theta, \delta, \rho)
    ab = model.create_attraction_basis(maxiter=maxiter, npts=npts)

    # Color map with colormesh
```

7.12. Globalization and Cycles
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()

fig = interact(interact_attraction_basis,
               \( \rho \in [0.0, 1.0, 0.05] \),
               maxiter=(50, 5000, 50),
               npts=(25, 750, 25))
These lectures look at important concepts in time series that are used in economics.

8.1 Covariance Stationary Processes

8.1.1 Overview

In this lecture we study covariance stationary linear stochastic processes, a class of models routinely used to study economic and financial time series.

This class has the advantage of being:

1. simple enough to be described by an elegant and comprehensive theory
2. relatively broad in terms of the kinds of dynamics it can represent

We consider these models in both the time and frequency domain.

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 processes can be represented in linear state space form
However, ARMA have some important structure that makes it valuable to study them separately

**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 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

**Other Reading**

For supplementary reading, see
- [LS18], chapter 2
- [Sar87], chapter 11
- John Cochrane notes on time series analysis, chapter 8
- [Shi95], chapter 6
- [CC08], all

**8.1.2 Introduction**

Consider a sequence of random variables \( \{X_t\} \) indexed by \( t \in \mathbb{Z} \) and taking values in \( \mathbb{R} \)

Thus, \( \{X_t\} \) begins in the infinite past and extends to the infinite future a convenient and standard assumption

As in other fields, successful economic modeling typically assumes the existence of features that are constant over time

If these assumptions are correct, then each new observation \( X_t, X_{t+1}, \ldots \) can provide additional information about the time-invariant features, allowing us to learn from as data arrive

For this reason, we will focus in what follows on processes that are stationary or become so after a transformation (see for example *this lecture* and *this lecture*)

**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 $\gamma: \mathbb{Z} \to \mathbb{R}$ 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.

**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. $\gamma(k) = \sigma^2 \mathbb{1}\{k = 0\}$ for some $\sigma > 0$ (Here $\mathbb{1}\{k = 0\}$ 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.

**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

$$X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}, \quad t \in \mathbb{Z} \tag{8.1}$$

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 (8.1) 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 (8.1) is

$$\gamma(k) = \sigma^2 \sum_{j=0}^{\infty} \psi_j \psi_{j+k} \tag{8.2}$$

By the Cauchy-Schwartz inequality one can show that $\gamma(k)$ satisfies equation (8.2).

Evidently, $\gamma(k)$ does not depend on $t$.

8.1. Covariance Stationary Processes
Wolds Decomposition

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
- \( \eta_t \) can be expressed as a linear function of \( X_{t-1}, X_{t-2}, \ldots \) and is perfectly predictable over arbitrarily long horizons.

For intuition and further discussion, see [Sar87], p. 286.

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

\[
X_t = \phi X_{t-1} + \epsilon_t \quad \text{where} \quad |\phi| < 1 \quad \text{and} \quad \{\epsilon_t\} \text{ is white noise}
\]  

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 (8.2) to the previous expression for \( X_t \), we get the AR(1) autocovariance function

\[
\gamma(k) = \phi^k \frac{\sigma^2}{1 - \phi^2}, \quad k = 0, 1, \ldots
\]  

The next figure plots an example of this function for \( \phi = 0.8 \) and \( \phi = -0.8 \) with \( \sigma = 1 \).

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

num_rows, num_cols = 2, 1
```
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

**ARMA Processes**

A stochastic process \{X_t\} is called an *autoregressive moving average process*, or ARMA(p,q), if it can be written as

\[
X_t = \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q}
\] (8.5)

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 (8.5) as

\[
L^0 X_t - \phi_1 L^1 X_t - \cdots - \phi_p L^p X_t = L^0 \epsilon_t + \theta_1 L^1 \epsilon_t + \cdots + \theta_q L^q \epsilon_t
\] (8.6)

If we let \(\phi(z)\) and \(\theta(z)\) be the polynomials

\[
\phi(z) := 1 - \phi_1 z - \cdots - \phi_p z^p \quad \text{and} \quad \theta(z) := 1 + \theta_1 z + \cdots + \theta_q z^q
\] (8.7)

then (8.6) becomes

\[
\phi(L) X_t = \theta(L) \epsilon_t
\] (8.8)

In what follows we **always assume** that the roots of the polynomial \(\phi(z)\) lie outside the unit circle in the complex plane.

This condition is sufficient to guarantee that the ARMA(p,q) process is covariance stationary.

In fact it implies that the process falls within the class of general linear processes *described above*.

That is, given an ARMA(p,q) process \{X_t\} satisfying the unit circle condition, there exists a square summable sequence \{\psi_t\} with \(X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}\) for all \(t\).

The sequence \{\psi_t\} can be obtained by a recursive procedure outlined on page 79 of [CC08].

The function \(t \mapsto \psi_t\) is often called the *impulse response function*.
8.1.3 Spectral Analysis

Autocovariance functions provide a great deal of information about covariance stationary processes.

In fact, for zero-mean Gaussian processes, the autocovariance function characterizes the entire joint distribution.

Even for non-Gaussian processes, it provides a significant amount of information.

It turns out that there is an alternative representation of the autocovariance function of a covariance stationary process, called the spectral density.

At times, the spectral density is easier to derive, easier to manipulate, and provides additional intuition.

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 \((xu - yv, xv + yu)\), 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 \((xu - yv, xv + yu)\) 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 \(\omega = \arctan(y/x)\) or \(\tan(\omega) = y/x\).

Spectral Densities

Let \(\{X_t\}\) be a covariance stationary process with autocovariance function \(\gamma\) satisfying \(\sum_k \gamma(k)^2 < \infty\).
The spectral density $f$ of $\{X_t\}$ is defined as the discrete time Fourier transform of its autocovariance function $\gamma$

$$f(\omega) := \sum_{k \in \mathbb{Z}} \gamma(k) e^{-i \omega k}, \quad \omega \in \mathbb{R}$$

(Some authors normalize the expression on the right by constants such as $1/\pi$ the convention chosen makes little difference provided you are consistent)

Using the fact that $\gamma$ is even, in the sense that $\gamma(t) = \gamma(-t)$ for all $t$, we can show that

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k)$$  \hspace{1cm} (8.9)

It is not difficult to confirm that $f$ is

- real-valued
- even ($f(\omega) = f(-\omega)$), and
- $2\pi$-periodic, in the sense that $f(2\pi + \omega) = f(\omega)$ for all $\omega$

It follows that the values of $f$ on $[0, \pi]$ determine the values of $f$ on all of $\mathbb{R}$ the proof is an exercise

For this reason it is standard to plot the spectral density only on the interval $[0, \pi]$

**Example 1: White Noise**

Consider a white noise process $\{\epsilon_t\}$ with standard deviation $\sigma$

It is easy to check that in this case $f(\omega) = \sigma^2$. So $f$ is a constant function

As we will see, this can be interpreted as meaning that all frequencies are equally present

(White light has this property when frequency refers to the visible spectrum, a connection that provides the origins of the term white noise)

**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 spectral density

$$f(\omega) = \sigma^2 \left(1 + 2\theta \cos(\omega) + \theta^2\right)$$  \hspace{1cm} (8.10)

With a bit more effort, it is possible to show (see, e.g., p. 261 of [Sar87]) that the spectral density of the AR(1) process $X_t = \phi X_{t-1} + \epsilon_t$ is

$$f(\omega) = \frac{\sigma^2}{1 - 2\phi \cos(\omega) + \phi^2}$$  \hspace{1cm} (8.11)
More generally, it can be shown that the spectral density of the ARMA process (8.5) is

\[ f(\omega) = \frac{\theta(e^{i\omega})}{\phi(e^{i\omega})} \sigma^2 \]  

(8.12)

where

- \( \sigma \) is the standard deviation of the white noise process \( \{\epsilon_t\} \)
- the polynomials \( \phi(\cdot) \) and \( \theta(\cdot) \) are as defined in (8.7)

The derivation of (8.12) uses the fact that convolutions become products under Fourier transformations.

The proof is elegant and can be found in many places, see, for example, [Sar87], chapter 11, section 4.

It's a nice exercise to verify that (8.10) and (8.11) are indeed special cases of (8.12).

**Interpreting the Spectral Density**

Plotting (8.11) reveals the shape of the spectral density for the AR(1) model when \( \phi \) takes the values 0.8 and -0.8 respectively.

```python
def ar1_sd(\omega):
    return 1 / (1 - 2 * np.cos(\omega) + ...)

\omega = 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, (phi) in enumerate((0.8, -0.8)):
    ax = axes[i]
    sd = ar1_sd(\omega)
    ax.plot(\omega, sd, 'b-', alpha=0.6, lw=2, label='spectral density, $\phi = \theta(\cdot)$')
    ax.legend(loc='upper center')
    ax.set(xlabel='frequency', xlim=(0, np.pi))
plt.show()```

8.1. Covariance Stationary Processes 987
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

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k) = \gamma(0) + 2 \sum_{k \geq 1} (-0.8)^k \cos(\omega k)$$  \hspace{1cm} (8.13)

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 (8.13) is large.

These ideas are illustrated in the next figure, which has $k$ on the horizontal axis.
\[
= -0.8
\]
\[
times = \text{list(range(16))}
\]
\[
y1 = [**k / (1 - **2) \text{ for } k \text{ in } \times]
\]
\[
y2 = \text{np.cos(np.pi * k) \text{ for } k \text{ in } \times}
\]
\[
y3 = [a \ast b \text{ for } a, b \text{ in } \text{zip}(y1, y2)]
\]

\[
\text{num_rows, num_cols = 3, 1}
\]
\[
\text{fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 8))}
\]
\[
\text{plt.subplots_adjust(hspace=0.25)}
\]
\[
\text{# Autocovariance when } = -0.8
\]
\[
\text{ax = axes[0]}
\]
\[
\text{ax.plot(times, y1, 'bo-', alpha=0.6, label='$$\gamma(k)$$')}
\]
\[
\text{ax.legend(loc='upper right')}
\]
\[
\text{ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)}
\]
\[
\text{# Cycles at frequence } \pi
\]
\[
\text{ax = axes[1]}
\]
\[
\text{ax.plot(times, y2, 'bo-', alpha=0.6, label='$$\cos(\pi k)$$')}
\]
\[
\text{ax.legend(loc='upper right')}
\]
\[
\text{ax.hlines(0, 0, 15, yticks=(-1, 0, 1))}
\]
\[
\text{# Product}
\]
\[
\text{ax = axes[2]}
\]
\[
\text{ax.stem(times, y3, label='$$\gamma(k) \cos(\pi k)$$')}
\]
\[
\text{ax.legend(loc='upper right')}
\]
\[
\text{ax.set(xlim=(0, 15), ylim=(-3, 3), yticks=(-1, 0, 1, 2, 3))}
\]
\[
\text{ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)}
\]
\[
\text{plt.show()}
\]
On the other hand, if we evaluate $f(\omega)$ at $\omega = \pi/3$, then the cycles are not matched, the sequence $\gamma(k) \cos(\omega k)$ contains both positive and negative terms, and hence the sum of these terms is much smaller.

```python
= -0.8
times = list(range(16))
y1 = [**k / (1 - **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)
```
In summary, the spectral density is large at frequencies $\omega$ where the autocovariance function exhibits damped cycles.

### 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

$$\gamma(k) = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(\omega)e^{i\omega k}d\omega$$

(8.14)

This is convenient in situations where the spectral density is easier to calculate and manipulate than the autocovariance function.

(For example, the expression (8.12) for the ARMA spectral density is much easier to work with than the expression for the ARMA autocovariance)

**Mathematical Theory**

This section is loosely based on [Sar87], p. 249-253, and included for those who

- would like a bit more insight into spectral densities
- and have at least some background in Hilbert space theory

Others should feel free to skip to the next section; none of this material is necessary to progress to computation.

Recall that every separable Hilbert space $H$ has a countable orthonormal basis $\{h_k\}$

The nice thing about such a basis is that every $f \in H$ satisfies

$$f = \sum_k \alpha_k h_k \quad \text{where} \quad \alpha_k := \langle f, h_k \rangle$$

(8.15)

where $\langle \cdot, \cdot \rangle$ denotes the inner product in $H$

Thus, $f$ can be represented to any degree of precision by linearly combining basis vectors.

The scalar sequence $\alpha = \{\alpha_k\}$ is called the *Fourier coefficients* of $f$, and satisfies $\sum_k |\alpha_k|^2 < \infty$

In other words, $\alpha$ is in $\ell_2$, the set of square summable sequences.

Consider an operator $T$ that maps $\alpha \in \ell_2$ into its expansion $\sum_k \alpha_k h_k \in H$.

The Fourier coefficients of $T\alpha$ are just $\alpha = \{\alpha_k\}$, as you can verify by confirming that $\langle T\alpha, h_k \rangle = \alpha_k$

Using elementary results from Hilbert space theory, it can be shown that

- $T$ is one-to-one if $\alpha$ and $\beta$ are distinct in $\ell_2$, then so are their expansions in $H$
- $T$ is onto if $f \in H$ then its preimage in $\ell_2$ is the sequence $\alpha$ given by $\alpha_k = \langle f, h_k \rangle$
- $T$ is a linear isometry in particular $\langle \alpha, \beta \rangle = \langle T\alpha, T\beta \rangle$
Summarizing these results, we say that any separable Hilbert space is isometrically isomorphic to \( \ell_2 \).

In essence, this says that each separable Hilbert space we consider is just a different way of looking at the fundamental space \( \ell_2 \).

With this in mind, let’s specialize to a setting where

- \( \gamma \in \ell_2 \) is the autocovariance function of a covariance stationary process, and \( f \) is the spectral density
- \( H = L_2 \), where \( L_2 \) is the set of square summable functions on the interval \([-\pi, \pi]\), with inner product \( \langle g, h \rangle = \int_{-\pi}^{\pi} g(\omega)h(\omega) d\omega \)
- \( \{h_k\} \) is the orthonormal basis for \( L_2 \) given by the set of trigonometric functions

\[
h_k(\omega) = \frac{e^{ik\omega}}{\sqrt{2\pi}}, \quad k \in \mathbb{Z}, \quad \omega \in [-\pi, \pi]
\]

Using the definition of \( T \) from above and the fact that \( f \) is even, we now have

\[
T\gamma = \sum_{k \in \mathbb{Z}} \gamma(k) \frac{e^{ik\omega}}{\sqrt{2\pi}} = \frac{1}{\sqrt{2\pi}} f(\omega)
\]

In other words, apart from a scalar multiple, the spectral density is just an transformation of \( \gamma \in \ell_2 \) under a certain linear isometry a different way to view \( \gamma \).

In particular, it is an expansion of the autocovariance function with respect to the trigonometric basis functions in \( L_2 \)

As discussed above, the Fourier coefficients of \( T\gamma \) are given by the sequence \( \gamma \), and, in particular, \( \gamma(k) = \langle T\gamma, h_k \rangle \)

Transforming this inner product into its integral expression and using (8.16) gives (8.14), justifying our earlier expression for the inverse transform.

### 8.1.4 Implementation

Most code for working with covariance stationary models deals with ARMA models.

Python code for studying ARMA models can be found in the `tsa` submodule of `statsmodels`.

Since this code doesn’t quite cover our needs particularly vis-a-vis spectral analysis we’ve put together the module `arma.py`, which is part of `QuantEcon.py` package.

The module provides functions for mapping ARMA(\( p, q \)) models into their

1. impulse response function
2. simulated time series
3. autocovariance function
4. spectral density
Application

Let's use this code to replicate the plots on pages 68–69 of [LS18].

Here are some functions to generate the plots:

```python
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,
            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,
            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()):
        ax
```

---

*QuantEcon.lectures-python3 PDF, Release 2018-Aug-8*
Now lets call these functions to generate the plots

Well use the model $X_t = 0.5X_{t-1} + \epsilon_t - 0.8\epsilon_{t-2}$

```python
import quantecon as qe
q = 0.5
theta = 0, -0.8
arma = qe.ARMA((, theta)
quad_plot(arma)
```

**Explanation**

The call

```
arma = ARMA(, theta, sigma)
```

creates an instance `arma` that represents the ARMA($p, q$) model

$$X_t = \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q}$$

If `phi` and `theta` are arrays or sequences, then the interpretation will be
• holds the vector of parameters \((\phi_1, \phi_2, \ldots, \phi_p)\)
• \(\theta\) holds the vector of parameters \((\theta_1, \theta_2, \ldots, \theta_q)\)

The parameter \(\sigma\) is always a scalar, the standard deviation of the white noise.

We also permit \(\phi\) and \(\theta\) 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 (8.8).

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

\[
times, \psi = \text{dimpulse}((\text{ma_poly}, \text{ar_poly}, 1), n=\text{impulse_length})
\]

where `ma_poly` and `ar_poly` correspond to the polynomials in (8.7) that is,

• `ma_poly` is the vector \((1, \theta_1, \theta_2, \ldots, \theta_q)\)
• `ar_poly` is the vector \((1, -\phi_1, -\phi_2, \ldots, -\phi_p)\)

To this end, we also maintain the arrays `ma_poly` and `ar_poly` as instance data, with their values computed automatically from the values of `phi` and `theta` supplied by the user.

If the user decides to change the value of either `theta` or `phi` ex-post by assignments such as `arma.phi = (0.5, 0.2)` or `arma.theta = (0, -0.1)`,

then `ma_poly` and `ar_poly` should update automatically to reflect these new parameters.

This is achieved in our implementation by using descriptors.

### 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 NumPys Fourier transform package `np.fft`, which wraps a standard Fortran-based package called FFTPACK.

A look at the `np.fft` documentation shows that the inverse transform `np.fft.ifft` takes a given sequence \(A_0, A_1, \ldots, A_{n-1}\) and returns the sequence \(a_0, a_1, \ldots, a_{n-1}\) defined by

\[
a_k = \frac{1}{n} \sum_{t=0}^{n-1} A_t e^{ik2\pi t/n}
\]
Thus, if we set \( A_t = f(\omega_t) \), where \( f \) is the spectral density and \( \omega_t := 2\pi t/n \), then

\[
a_k = \frac{1}{n} \sum_{t=0}^{n-1} f(\omega_t) e^{i\omega t k} = \frac{1}{n} \sum_{t=0}^{n-1} f(\omega_t) e^{i\omega t k}, \quad \omega_t := 2\pi t/n
\]

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 (8.14) we have now shown that, for \( n \) sufficiently large, \( a_k \approx \gamma(k) \) which is exactly what we want to compute

### 8.2 Estimation of Spectra

#### Contents

- Estimation of Spectra
  - Overview
  - Periodograms
  - Smoothing
  - Exercises
  - Solutions

#### 8.2.1 Overview

In a previous lecture we covered some fundamental properties of covariance stationary linear stochastic processes.

One objective for that lecture was to introduce spectral densities, a standard and very useful technique for analyzing such processes.

In this lecture we turn to the problem of estimating spectral densities and other related quantities from data.

Estimates of the spectral density are computed using what is known as a periodogram, which in turn is computed via the famous fast Fourier transform.

Once the basic technique has been explained, we will apply it to the analysis of several key macroeconomic time series.

For supplementary reading, see [Sar87] or [CC08].
8.2.2 Periodograms

Recall that the spectral density $f$ of a covariance stationary process with autocorrelation function $\gamma$ can be written

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k), \quad \omega \in \mathbb{R}$$

Now consider the problem of estimating the spectral density of a given time series, when $\gamma$ is unknown. In particular, let $X_0, \ldots, X_{n-1}$ be $n$ consecutive observations of a single time series that is assumed to be covariance stationary.

The most common estimator of the spectral density of this process is the periodogram of $X_0, \ldots, X_{n-1}$, which is defined as

$$I(\omega) := \frac{1}{n} \left| \sum_{t=0}^{n-1} X_t e^{i t \omega} \right|^2, \quad \omega \in \mathbb{R} \quad (8.17)$$

(Recall that $|z|$ denotes the modulus of complex number $z$)

Alternatively, $I(\omega)$ can be expressed as

$$I(\omega) = \frac{1}{n} \left\{ \left( \sum_{t=0}^{n-1} X_t \cos(\omega t) \right)^2 + \left( \sum_{t=0}^{n-1} X_t \sin(\omega t) \right)^2 \right\}$$

It is straightforward to show that the function $I$ is even and $2\pi$-periodic (i.e., $I(\omega) = I(-\omega)$ and $I(\omega + 2\pi) = I(\omega)$ for all $\omega \in \mathbb{R}$).

From these two results, you will be able to verify that the values of $I$ on $[0, \pi]$ determine the values of $I$ on all of $\mathbb{R}$.

The next section helps to explain the connection between the periodogram and the spectral density.

**Interpretation**

To interpret the periodogram, it is convenient to focus on its values at the Fourier frequencies

$$\omega_j := \frac{2\pi j}{n}, \quad j = 0, \ldots, n - 1$$

In what sense is $I(\omega_j)$ an estimate of $f(\omega_j)$?

The answer is straightforward, although it does involve some algebra.

With a bit of effort one can show that, for any integer $j > 0$,

$$\sum_{t=0}^{n-1} e^{i t \omega_j} = \sum_{t=0}^{n-1} \exp \left\{ i 2\pi j \frac{t}{n} \right\} = 0$$
Letting $\bar{X}$ denote the sample mean $n^{-1} \sum_{t=0}^{n-1} X_t$, we then have

$$n I(\omega_j) = \left| \sum_{t=0}^{n-1} (X_t - \bar{X}) e^{it\omega_j} \right|^2 = \sum_{t=0}^{n-1} (X_t - \bar{X}) e^{it\omega_j} \sum_{r=0}^{n-1} (X_r - \bar{X}) e^{-ir\omega_j}$$

By carefully working through the sums, one can transform this to

$$n I(\omega_j) = \sum_{t=0}^{n-1} (X_t - \bar{X})^2 + 2 \sum_{k=1}^{n-1} \sum_{t=k}^{n-1} (X_t - \bar{X}) (X_{t-k} - \bar{X}) \cos(\omega_j k)$$

Now let

$$\hat{\gamma}(k) := \frac{1}{n} \sum_{t=k}^{n-1} (X_t - \bar{X}) (X_{t-k} - \bar{X}), \quad k = 0, 1, \ldots, n - 1$$

This is the sample autocovariance function, the natural plug-in estimator of the autocovariance function $\gamma$ (Plug-in estimator is an informal term for an estimator found by replacing expectations with sample means)

With this notation, we can now write

$$I(\omega_j) = \hat{\gamma}(0) + 2 \sum_{k=1}^{n-1} \hat{\gamma}(k) \cos(\omega_j k)$$

Recalling our expression for $f$ given above, we see that $I(\omega_j)$ is just a sample analog of $f(\omega_j)$

**Calculation**

Let's now consider how to compute the periodogram as defined in (8.17)

There are already functions available that will do this for us an example is `statsmodels.tsa.stattools.periodogram` in the `statsmodels` package.

However, it is very simple to replicate their results, and this will give us a platform to make useful extensions.

The most common way to calculate the periodogram is via the discrete Fourier transform, which in turn is implemented through the fast Fourier transform algorithm.

In general, given a sequence $a_0, \ldots, a_{n-1}$, the discrete Fourier transform computes the sequence

$$A_j := \sum_{t=0}^{n-1} a_t \exp \left\{ i 2\pi \frac{tj}{n} \right\}, \quad j = 0, \ldots, n - 1$$

With `numpy.fft.fft` imported as `fft` and $a_0, \ldots, a_{n-1}$ stored in NumPy array $a$, the function call `fft(a)` returns the values $A_0, \ldots, A_{n-1}$ as a NumPy array

It follows that, when the data $X_0, \ldots, X_{n-1}$ are stored in array $X$, the values $I(\omega_j)$ at the Fourier frequencies, which are given by

$$\frac{1}{n} \sum_{t=0}^{n-1} X_t \exp \left\{ i 2\pi \frac{tj}{n} \right\}^2, \quad j = 0, \ldots, n - 1$$
can be computed by \( \text{np.abs}(\text{fft}(X))**2 / \text{len}(X) \)

Note: The NumPy function \text{abs} acts elementwise, and correctly handles complex numbers (by computing their modulus, which is exactly what we need)

A function called \text{periodogram} that puts all this together can be found \text{here}

Lets generate some data for this function using the \text{ARMA} class from \text{QuantEcon.py} (see the \text{lecture on linear processes} for more details)

Heres a code snippet that, once the preceding code has been run, generates data from the process

\[
X_t = 0.5X_{t-1} + \epsilon_t - 0.8\epsilon_{t-2}
\]  \hspace{1cm} (8.18)

where \( \{\epsilon_t\} \) is white noise with unit variance, and compares the periodogram to the actual spectral density

```python
import matplotlib.pyplot as plt
from quantecon import ARMA, periodogram

n = 40 # Data size
\( \theta = 0.5, (0, -0.8) \) # AR and MA parameters
lp = ARMA(\( \theta \))
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()
```

Running this should produce a figure similar to this one
This estimate looks rather disappointing, but the data size is only 40, so perhaps its 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

### 8.2.3 Smoothing

There are two related issues here

One is that, given the way the fast Fourier transform is implemented, the number of points $\omega$ at which $I(\omega)$ is estimated increases in line with the amount of data

In other words, although we have more data, we are also using it to estimate more values

A second issue is that densities of all types are fundamentally hard to estimate without parametric assumptions

Typically, nonparametric estimation of densities requires some degree of smoothing

The standard way that smoothing is applied to periodograms is by taking local averages
In other words, the value $I(\omega_j)$ is replaced with a weighted average of the adjacent values

$$I(\omega_{j-p}), I(\omega_{j-p+1}), \ldots, I(\omega_j), \ldots, I(\omega_{j+p})$$

This weighted average can be written as

$$I_S(\omega_j) := \sum_{\ell=-p}^{p} w(\ell) I(\omega_{j+\ell}) \quad (8.19)$$

where the weights $w(-p), \ldots, w(p)$ are a sequence of $2p + 1$ nonnegative values summing to one

In generally, 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 (8.19)

```python
import numpy as np
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)
plt.figure(figsize=(9, 7))
plt.plot(x, window)
plt.title("Hanning window")
plt.ylabel("Weights")
plt.xlabel("Position in sequence of weights")
plt.show()
```

8.2. Estimation of Spectra
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 (8.18) and there are 400 observations.)

From 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.

8.2. Estimation of Spectra
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.

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 (8.19)

The underlying assumption that makes this a good idea is that the true spectral density is relatively regular: the value of $I(\omega)$ is close to that of $I(\omega')$ when $\omega$ is close to $\omega'$.

This will not be true in all cases, but it is certainly true for white noise.

For white noise, $I$ is as regular as possible: it is a constant function.

In this case, values of $I(\omega')$ at points $\omega'$ near to $\omega$ provided the maximum possible amount of information about the value $I(\omega)$.

Another way to put this is that if $I$ is relatively constant, then we can use a large amount of smoothing without introducing too much bias.

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

\[
X_{t+1} = \mu + \phi X_t + \epsilon_{t+1}
\]

(8.20)
where $\mu$ and $\phi \in (-1, 1)$ are unknown parameters and $\{\epsilon_t\}$ is white noise.

It follows that if we regress $X_{t+1}$ on $X_t$ and an intercept, the residuals will approximate white noise. Let

- $g$ be the spectral density of $\{\epsilon_t\}$ a constant function, as discussed above
- $I_0$ be the periodogram estimated from the residuals an estimate of $g$
- $f$ be the spectral density of $\{X_t\}$ the object we are trying to estimate

In view of an earlier result we obtained while discussing ARMA processes, $f$ and $g$ are related by

$$f(\omega) = \left| \frac{1}{1 - \phi e^{i\omega}} \right|^2 g(\omega)$$

(8.21)

This suggests that the recoloring step, which constructs an estimate $I$ of $f$ from $I_0$, should set

$$I(\omega) = \left| \frac{1}{1 - \hat{\phi} e^{i\omega}} \right|^2 I_0(\omega)$$

where $\hat{\phi}$ is the OLS estimate of $\phi$.

The code for `ar_periodogram()` the third function in `estspec.py` does exactly this. (See the code here)

The next figure shows realizations of the two kinds of smoothed periodograms

1. standard smoothed periodogram, the ordinary smoothed periodogram, and
2. AR smoothed periodogram, the pre-whitened and recolored one generated by `ar_periodogram()`

The periodograms are calculated from time series drawn from (8.20) 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

8.2. Estimation of Spectra
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.
8.2.4 Exercises

Exercise 1

Replicate this figure (modulo randomness)

The model is as in equation (8.18) and there are 400 observations

For the smoothed periodogram, the window type is hamming

Exercise 2

Replicate this figure (modulo randomness)

The model is as in equation (8.20), 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

Exercise 3

To be written. The exercise will be to use the code from this lecture to download FRED data and generate periodograms for different kinds of macroeconomic data.

8.2.5 Solutions

```python
from quantecon import ar_periodogram

Exercise 1

## Data
n = 400
= 0.5
= 0, -0.8
lp = ARMA((, )
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')
```

8.2. Estimation of Spectra
```python
ax[i].legend()
ax[i].set_title(f'window length = {wl}')
plt.show()
```
Exercise 2

lp = ARMA(-0.9)
w1 = 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=w1)
    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=w1)
    ax[i].semilogy(x, y_ar, 'b-', lw=2, alpha=0.75, label='AR smoothed periodogram')

    ax[i].legend(loc='upper left')
plt.show()
8.3 Additive Functionals
8.3.1 Overview

Some time series are nonstationary. For example, output, prices, and dividends are typically nonstationary, due to irregular but persistent growth. Which kinds of models are useful for studying such time series? Hansen and Scheinkman [HS09] analyze 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. For example, 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 [HS17] (chs. 5 and 6) describe discrete time versions of additive and multiplicative functionals.

In this lecture we discuss the former (i.e., additive functionals). In the next lecture we discuss multiplicative functionals. We also consider fruitful decompositions of additive and multiplicative processes, a more in-depth discussion of which can be found in Hansen and Sargent [HS17].

8.3.2 A Particular Additive Functional

This lecture focuses on a particular type of additive functional: a scalar process \( \{y_t\}_{t=0}^{\infty} \) whose increments are driven by a Gaussian vector autoregression.

It is simple to construct, simulate, and analyze. This additive functional consists of two components, the first of which is a **first-order vector autoregression** (VAR).
\[ x_{t+1} = Ax_t + Bz_{t+1} \]  
(8.22)

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),
- \( z_{t+1} \sim \mathcal{N}(0, I) \) is an \( m \times 1 \) i.i.d. shock,
- \( B \) is an \( n \times m \) matrix, and
- \( x_0 \sim \mathcal{N}(\mu_0, \Sigma_0) \) is a random initial condition for \( x \)

The second component is an equation that expresses increments of \( \{y_t\}_{t=0}^{\infty} \) as linear functions of

- a scalar constant \( \nu \),
- the vector \( x_t \), and
- the same Gaussian vector \( z_{t+1} \) that appears in the VAR (8.22)

In particular,

\[ y_{t+1} - y_t = \nu + Dx_t + Fz_{t+1} \]  
(8.23)

Here \( y_0 \sim \mathcal{N}(\mu_y, \Sigma_y) \) is a random initial condition

The nonstationary random process \( \{y_t\}_{t=0}^{\infty} \) displays systematic but random *arithmetic growth*

**A linear state space representation**

One way to represent the overall dynamics is to use a *linear state space system*

To do this, we set up state and observation vectors

\[ \hat{x}_t = \begin{bmatrix} 1 \\ x_t \\ y_t \end{bmatrix} \quad \text{and} \quad \hat{y}_t = \begin{bmatrix} x_t \\ y_t \end{bmatrix} \]

Now we construct the state space system

\[
\begin{bmatrix}
1 \\
\hat{x}_{t+1} \\
y_{t+1}
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 \\
0 & A & 0 \\
\nu & D' & 1
\end{bmatrix}
\begin{bmatrix}
1 \\
x_t \\
y_t
\end{bmatrix} +
\begin{bmatrix}
0 \\
B \\
F'
\end{bmatrix}
\begin{bmatrix}
z_{t+1}
\end{bmatrix}
\]

\[
\begin{bmatrix}
x_t \\
y_t
\end{bmatrix} =
\begin{bmatrix}
0 & I & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 \\
x_t \\
y_t
\end{bmatrix}
\]

1014 Chapter 8. Time Series Models
This can be written as
\[
\begin{align*}
\tilde{x}_{t+1} &= \tilde{A}\tilde{x}_t + \tilde{B}z_{t+1} \\
\tilde{y}_t &= \tilde{D}\tilde{x}_t
\end{align*}
\]
which is a standard linear state space system.

To study it, we could map it into an instance of `LinearStateSpace` from `QuantEcon.py`.

We will in fact use a different set of code for simulation, for reasons described below.

### 8.3.3 Dynamics

Let’s run some simulations to build intuition.

In doing so well assume that $z_{t+1}$ is scalar and that $\tilde{x}_t$ follows a 4th-order scalar autoregression

\[
\begin{align*}
\tilde{x}_{t+1} &= \phi_1 \tilde{x}_t + \phi_2 \tilde{x}_{t-1} + \phi_3 \tilde{x}_{t-2} + \phi_4 \tilde{x}_{t-3} + \sigma z_{t+1} 
\end{align*}
\]

(8.24)

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 (8.24) is not a first order system like (8.22), 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 (8.22) – (8.23) by appropriate selection of the matrices $A, B, D, F$.

You can try writing these matrices down now as an exercise the correct expressions will appear in the code below.

### Simulation

When simulating we embed our variables into a bigger system.

This system also constructs the components of the decompositions of $y_t$ and of $\exp(y_t)$ proposed by Hansen and Scheinkman [HS09].

All of these objects are computed using the code below:

```python
"""
@authors: Chase Coleman, Balint Szoke, Tom Sargent
"""
import numpy as np
import scipy as sp
```

8.3. Additive Functionals
import scipy.linalg as la
import quantecon as qe
import matplotlib.pyplot as plt
from scipy.stats import norm, lognorm

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
    if self.add_decomp:
        v, H, g = self.add_decomp
    else:
        v, 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
    A3 = np.hstack([nx0c, nx0c, A, nyx0m.T, nyx0m.T])  # Transition for
    # Transition for
    A4 = np.hstack([v, ny0c, D, ny1m, ny0m])  # Transition for
    # Transition for
    A5 = np.hstack([ny0c, ny0c, nyx0m, ny0m, ny0m])  # Transition for
    # Transition for
    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]
    G1 = np.hstack([nx0c, nx0c, np.eye(nx), nyx0m.T, nyx0m.T])  #
    G2 = np.hstack([ny0c, ny0c, nyx0m, ny1m, ny0m])  #
    G3 = np.hstack([ny0c, ny0c, nyx0m, ny0m, ny1m])  #
    G4 = np.hstack([ny0c, ny0c, -g, ny0m, ny0m])  #
    G5 = np.hstack([ny0c, v, nyx0m, ny0m, ny0m])  #
    Gbar = np.vstack([G1, G2, G3, G4, G5])

8.3. Additive Functionals
# Build H matrix for LSS
\[ H_{\text{bar}} = \text{np.zeros}((G_{\text{bar}}.\text{shape}[0], \text{nk})) \]

# Build LSS type
\[ x_0 = \text{np.hstack([1, 0, nx0r, ny0r, ny0r])} \]
\[ S_0 = \text{np.zeros}((\text{len}(x_0), \text{len}(x_0))) \]
\[ \text{lss} = \text{qe.lss.LinearStateSpace}(A_{\text{bar}}, B_{\text{bar}}, G_{\text{bar}}, H_{\text{bar}}, \mu_0=x_0, \Sigma_0=S_0) \]

def additive_decomp(self):
    """
    Return values for the martingale decomposition
    - \( \nu \) : unconditional mean difference in \( Y \)
    - \( H \) : coefficient for the (linear) martingale component
    - \( g \) : 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 = \text{np.identity}(self.nx)
    A_res = \text{la.solve}(I - self.A, I)
    g = self.D @ A_res
    H = self.F + self.D @ A_res @ self.B

    return self.\( \nu \), H, g

def multiplicative_decomp(self):
    """
    Return values for the multiplicative decomposition (Example 5.4.4.)
    - \( \nu_{\text{tilde}} \) : eigenvalue
    - \( H \) : vector for the Jensen term
    """
    \( \nu, H, g = self.\text{additive_decomp()} \)
    \( \nu_{\text{tilde}} = \nu + (0.5)*\text{np.expand_dims}(\text{np.diag}(H @ H.T), 1) \)

    return \( \nu_{\text{tilde}} \), H, g

def loglikelihood_path(self, x, y):
    k, T = y.shape
    FF = F @ F.T
    FFinv = \text{la.inv}(FF)
    temp = y[:, 1:] - y[:, :-1] - D @ x[:, :-1]
    obs = temp * FFinv * temp
    obssum = \text{np.cumsum}(obs)
    scalar = (\text{np.log}(\text{la.det}(FF)) + k*\text{np.log}(2*\text{np.pi}))*\text{np.arange}(1, T)

    return -(0.5)*(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
      _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,]
                                 t*(.5)*np.expand_dims(np.
                                 diag(H @ H.T),1)[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,]
                                   t*-.5)*np.expand_dims(np.
                                 diag(H @ H.T),1)[ii])),
              scale = np.asscalar( np.exp(-
                                  ymeans[nx+2+nm+ii])))

1020 Chapter 8. Time Series Models
# Pull out paths

```python
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.
        range(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.
        range(T)*(.5)*np.expand_dims(np.diag(H @ H.T),1)[ii]
```

```python
mult_figs = []
for ii in range(nm):
    li, ui = npaths*(ii), npaths*(ii+1)
    LI, UI = 2*(ii), 2*(ii+1)
    mpath_mult, mbounds_mult = np.empty((nm*tuple((npaths, T)))),
    mbounds_mult, sbounds_mult = np.empty((nm*2, T)),
    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[L:U,:], sbounds_mult[L:U,:], 1,
    show_trend=show_trend))
    mult_figs[ii].supertitle(f'Multiplicative decomposition of $y_{ii+1}$
    $', fontsize=14)
return mult_figs
```

```python
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
    v_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
```

8.3. Additive Functionals
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.
->range(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)
    mpath_mult[li:ui, :],
    mbounds_mult[LI:UI, :],
    horline=1)
    mart_figs[ii].suptitle(f'Martingale components for many paths of
->Sy_{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")
For now, we just plot $y_t$ and $x_t$, postponing until later a description of exactly how we compute them.

```python
_1, _2, _3, _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([[_1, _2, _3, _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('A particular 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 $y_t$

**Decomposition**

Hansen and Sargent \cite{HS17} 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 (8.22) and (8.23), we
first construct the matrices 

\[ H := F + B'(I - A')^{-1} D \]

\[ g := D'(I - A)^{-1} \]

Then the Hansen-Scheinkman [HS09] decomposition is

\[ y_t = \begin{cases} 
\text{trend component} & t \\
\text{Martingale component} & \sum_{j=1}^{t} H z_j \\
\text{stationary component} & \sum_{j=1}^{t} g x_j \\
\text{initial conditions} & g x_0 + y_0 
\end{cases} \]

At this stage you should pause and verify that \( y_{t+1} - y_t \) satisfies (8.23)

It is convenient for us to introduce the following notation:

- \( \tau_t = \nu t \), a linear, deterministic trend
- \( m_t = \sum_{j=1}^{t} H z_j \), a martingale with time \( t + 1 \) increment \( H z_{t+1} \)
- \( s_t = g x_t \), an (asymptotically) stationary component

We want to characterize and simulate components \( \tau_t, m_t, s_t \) of the decomposition.

A convenient way to do this is to construct an appropriate instance of a linear state space system by using \texttt{LinearStateSpace} from \texttt{QuantEcon.py}

This will allow us to use the routines in \texttt{LinearStateSpace} to study dynamics

To start, observe that, under the dynamics in (8.22) and (8.23) and with the definitions just given,

\[
\begin{bmatrix}
1 \\
t+1 \\
x_{t+1} \\
y_{t+1} \\
m_{t+1}
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 0 & A & 0 & 0 \\
\nu & 0 & D' & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 \\
x_t \\
y_t \\
\tau_t \\
m_t \\
s_t
\end{bmatrix} +
\begin{bmatrix}
0 \\
0 \\
B \\
F' \\
H'
\end{bmatrix} z_{t+1}
\]

and

\[
\begin{bmatrix}
x_t \\
y_t \\
\tau_t \\
m_t \\
s_t
\end{bmatrix} =
\begin{bmatrix}
0 & 0 & I & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & \nu & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & -g & 0 & 0
\end{bmatrix}
\begin{bmatrix}
1 \\
t \\
x_t \\
y_t \\
\tau_t \\
m_t \\
s_t
\end{bmatrix}
\]

With

\[
\tilde{x} := \begin{bmatrix}
1 \\
x_t \\
y_t \\
\tau_t \\
m_t
\end{bmatrix} \quad \text{and} \quad \tilde{y} := \begin{bmatrix}
x_t \\
y_t \\
\tau_t \\
m_t \\
s_t
\end{bmatrix}
\]

we can write this as the linear state space system

\[ \tilde{x}_{t+1} = \tilde{A} \tilde{x}_t + \tilde{B} z_{t+1} \]
\[ \tilde{y}_t = \tilde{D} \tilde{x}_t \]

By picking out components of \( \tilde{y}_t \), we can track all variables of interest
8.3.4 Code

The class AMF_LSS_VAR mentioned above does all that we want to study our additive functional

In fact AMF_LSS_VAR does more, as we shall explain below

(A hint that it does more is the name of the class – here AMF stands for additive and multiplicative functional – the code will do things for 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

```python
amf.plot_additive(T)
plt.show()
```

You will generate (modulo randomness) the plot

When we plot multiple realizations of a component in the 2nd, 3rd, and 4th panels, we also plot population 95% probability coverage sets computed using the LinearStateSpace class

We have chosen to simulate many paths, all starting from the same nonrandom 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
An 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 the multiplicative decomposition of the \( M_t \)

\[
\frac{M_t}{M_0} = \exp(t\nu) \exp \left( \sum_{j=1}^{t} H \cdot Z_j \right) \exp \left( D'(I - A)^{-1} x_0 - D'(I - A)^{-1} x_t \right)
\]

or

\[
\frac{M_t}{M_0} = \exp (\tilde{\nu} t) \left( \frac{\tilde{M}_t}{M_0} \right) \left( \frac{\tilde{e}(x_0)}{\tilde{e}(x_t)} \right)
\]

where

\[
\tilde{\nu} = \nu + \frac{H \cdot H}{2}, \quad \tilde{M}_t = \exp \left( \sum_{j=1}^{t} \left( H \cdot z_j - \frac{H \cdot H}{2} \right) \right), \quad \tilde{M}_0 = 1
\]

and

\[
\tilde{e}(x) = \exp[g(x)] = \exp[D'(I - A)^{-1} x]
\]

An instance of class `AMF_LSS_VAR` 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

```python
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.

**A peculiar large sample property**

Hansen and Sargent \(HS17\) (ch. 6) note that the martingale component \(\tilde{M}_t\) of the multiplicative decomposition has a peculiar property:

- While \(E_0\tilde{M}_t = 1\) for all \(t \geq 0\), nevertheless . . .
  - As \(t \rightarrow +\infty\), \(\tilde{M}_t\) converges to zero almost surely

The following simulation of many paths of \(\tilde{M}_t\) illustrates this property:

```python
np.random.seed(10021987)
amf.plot_martingales(12000)
plt.show()
```

Here’s the resulting figure:
8.4 Multiplicative Functionals

Contents

- Multiplicative Functionals
  - Overview
  - A Log-Likelihood Process
  - Benefits from Reduced Aggregate Fluctuations

Co-authors: Chase Coleman and Balint Szoke

8.4.1 Overview

This lecture is a sequel to the lecture on additive functionals

That lecture

1. defined a special class of additive functionals driven by a first-order vector VAR
2. by taking the exponential of that additive functional, created an associated multiplicative functional

This lecture uses this special class to create and analyze two examples

- A log likelihood process, an object at the foundation of both frequentist and Bayesian approaches to statistical inference
8.4.2 A Log-Likelihood Process

Consider a vector of additive functionals \( \{y_t\}_{t=0}^{\infty} \) described by

\[
\begin{align*}
  x_{t+1} &= Ax_t + Bz_{t+1} \\
  y_{t+1} - y_t &= Dx_t + Fz_{t+1},
\end{align*}
\]

where \( A \) is a stable matrix, \( \{z_{t+1}\}_{t=0}^{\infty} \) is an i.i.d. sequence of \( \mathcal{N}(0, I) \) random vectors, \( F \) is nonsingular, and \( x_0 \) and \( y_0 \) are vectors of known numbers.

Evidently,

\[
x_{t+1} = (A - BF^{-1}D)x_t + BF^{-1}(y_{t+1} - y_t),
\]

so that \( x_{t+1} \) can be constructed from observations on \( \{y_s\}_{s=0}^{t+1} \) and \( x_0 \).

The distribution of \( y_{t+1} - y_t \) conditional on \( x_t \) is normal with mean \( Dx_t \) and nonsingular covariance matrix \( FF' \).

Let \( \theta \) denote the vector of free parameters of the model.

These parameters pin down the elements of \( A, B, D, F \).

The log likelihood function of \( \{y_s\}_{s=1}^{t} \) is

\[
\log L_t(\theta) = -\frac{1}{2} \sum_{j=1}^{t} (y_j - y_{j-1} - Dx_{j-1})'(FF')^{-1}(y_j - y_{j-1} - Dx_{j-1}) - \frac{kt}{2} \log \det(FF') - \frac{kt}{2} \log(2\pi)
\]

Let's consider the case of a scalar process in which \( A, B, D, F \) are scalars and \( z_{t+1} \) is a scalar stochastic process.

We let \( \theta_o \) denote the true values of \( \theta \), meaning the values that generate the data.

For the purposes of this exercise, set \( \theta_o = (A, B, D, F) = (0.8, 1, 0.5, 0.2) \).

Set \( x_0 = y_0 = 0 \)

**Simulating sample paths**

Let's write a program to simulate sample paths of \( \{x_t, y_t\}_{t=0}^{\infty} \).

We do this by formulating the additive functional as a linear state space model and putting the LinearStateSpace class to work.
import numpy as np
import scipy as sp
import scipy.linalg as la
import quantecon as qe
import matplotlib.pyplot as plt
from scipy.stats import lognorm

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, \( \nu \)=0.0):
        # Unpack required elements
        self.A, self.B, self.D, self.F, self.\( \nu \) = A, B, D, F, \( \nu \)

        # 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, \( \nu \) = self.A, self.B, self.D, self.F, self.\( \nu \)
        nx, nk, nm = 1, 1, 1
        if self.add_decomp:
            \( \nu \), H, g = self.add_decomp
        else:
            \( \nu \), 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([\( \nu \), 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, 0, 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(x) : coefficient for the stationary component
    - 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

    return self.ν, H, g

def multiplicative_decomp(self):
    ""
    Return values for the multiplicative decomposition (Example 5.4.4.)
    ""
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`

```python
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
```

1034 Chapter 8. Time Series Models
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 tool kit, let’s apply them to run some simulations.

In particular, let’s use our program to generate \( I = 5000 \) sample paths of length \( T = 150 \), labeled \( \{x_t^i, y_t^i\}_{i=0}^{\infty} \) for \( i = 1, ..., I \).

Then we compute averages of \( \frac{1}{I} \sum_i x_t^i \) and \( \frac{1}{I} \sum_i y_t^i \) across the sample paths and compare them with the population means of \( x_t \) and \( y_t \).

Here goes

\[
A, B, D, F = [0.8, 1.0, 0.5, 0.2]
\]

amf = AMF_LSS_VAR(A, B, D, F=F)

T = 150
I = 5000

# Simulate and compute sample means
Xit, Yit = simulate_paths(amf, T, I)
Xmean_t = np.mean(Xit, 0)
Ymean_t = np.mean(Yit, 0)

# Compute population means
Xmean_pop, Ymean_pop = population_means(amf, T)

# Plot sample means vs population means
fig, ax = plt.subplots(2, figsize=(14, 8))

ax[0].plot(Xmean_t, label=r'$\frac{1}{I} \sum_i x_t^i$', color='b')
ax[0].plot(Xmean_pop, label='$\mathbb{E} x_t$', color='k')
ax[0].set_title('$x_t$')
ax[0].set_xlim((0, T))
ax[0].legend(loc=0)

ax[1].plot(Ymean_t, label=r'$\frac{1}{I} \sum_i y_t^i$', color='b')
ax[1].plot(Ymean_pop, label='$\mathbb{E} y_t$', color='k')
ax[1].set_title('$y_t$')
ax[1].set_xlim((0, T))
ax[1].legend(loc=0)
plt.show()

Here's the resulting figure

![Graph showing log-likelihoods](image)

**Simulating log-likelihoods**

Our next aim is to write a program to simulate \( \{\log L_t \mid \theta_o\}_{t=1}^T \).

We want as inputs to this program the same sample paths \( \{x^i_t, y^i_t\}_{t=0}^T \) that we have already computed.

We now want to simulate \( I = 5000 \) paths of \( \{\log L^i_t \mid \theta_o\}_{t=1}^T \):

- For each path, we compute \( \log L^i_T / T \)
- We also compute \( \frac{1}{T} \sum_{i=1}^I \log L^i_T / T \)

Then we to compare these objects.

Below we plot the histogram of \( \log L^i_T / T \) for realizations \( i = 1, \ldots, 5000 \)

```python
def simulate_likelihood(amf, Xit, Yit):
    # Get size
    I, T = Xit.shape

    # Allocate space
    LLit = np.empty((I, T-1))
```
```python
for i in range(I):
    LLit[i, :] = amf.loglikelihood_path(Xit[i, :], Yit[i, :])

return LLit

# Get likelihood from each path x^{i}, Y^{i}
LLit = simulate_likelihood(amf, Xit, Yit)

LLT = 1/T * LLit[:, -1]
LLmean_t = np.mean(LLT)

fig, ax = plt.subplots()

ax.hist(LLT)
ax.vlines(LLmean_t, ymin=0, ymax=I//3, color="k", linestyle="--", alpha=0.6)
plt.title(r"Distribution of $\frac{1}{T}\log L_{T} \mid \theta_{0}$")
plt.show()
```

Here's the resulting figure

![Distribution of \(\frac{1}{T}\log L_T \mid \theta_0\)](image)

Notice that the log likelihood is almost always nonnegative, implying that \(L_t\) is typically bigger than 1.

Recall that the likelihood function is a pdf (probability density function) and **not** a probability measure, so it can take values larger than 1.

In the current case, the conditional variance of \(\Delta y_{t+1}\), which equals \(FF^T = 0.04\), is so small that the maximum value of the pdf is 2 (see the figure below).
This implies that approximately 75% of the time (a bit more than one sigma deviation), we should expect the increment of the log likelihood to be nonnegative.

Let's see this in a simulation:

```python
normdist = sp.stats.norm(0, F)
mult = 1.175
print(f'The pdf at +/- {mult} sigma takes the value: {normdist.pdf(mult * F)}')
print(f'Probability of dL being larger than 1 is approx: {normdist.cdf(mult * F) - normdist.cdf(-mult * F)}')

# Compare this to the sample analogue:
L_increment = LLit[:, 1:] - LLit[:, :-1]
r, c = L_increment.shape
frac_nonegative = np.sum(L_increment >= 0) / (c * r)
print(f'Fraction of dlogL being nonnegative in the sample is: {frac_nonegative}')
```

Here's the output:

The pdf at +/- 1.175 sigma takes the value: 1.0001868966924388
Probability of dL being larger than 1 is approx: 0.7600052842019751
Fraction of dlogL being nonnegative in the sample is: 0.7601783783783784

Let's also plot the conditional pdf of $\Delta y_{t+1}$

```python
xgrid = np.linspace(-1, 1, 100)
plt.plot(xgrid, normdist.pdf(xgrid))
plt.title('Conditional pdf $f(\Delta y_{t+1} | x_t)$')
print(f'The pdf at +/- one sigma takes the value: {normdist.pdf(F)}')
plt.show()
```

Here's the resulting figure.
An alternative parameter vector

Now consider alternative parameter vector $\theta_1 = [A, B, D, F] = [0.9, 1.0, 0.55, 0.25]$

We want to compute $\{\log L_t \mid \theta_1\}_{t=1}^T$

The $x_t, y_t$ inputs to this program should be exactly the same sample paths $\{x_t^i, y_t^i\}_{t=0}^T$ that we computed above

This is because we want to generate data under the $\theta_0$ probability model but evaluate the likelihood under the $\theta_1$ model

So our task is to use our program to simulate $I = 5000$ paths of $\{\log L_t^i \mid \theta_1\}_{t=1}^T$

- For each path, compute $\frac{1}{T} \log L_T^i$
- Then compute $\frac{1}{I} \sum_{i=1}^I \frac{1}{T} \log L_T^i$

We want to compare these objects with each other and with the analogous objects that we computed above

Then we want to interpret outcomes

A function that we constructed can handle these tasks

The only innovation is that we must create an alternative model to feed in

We will creatively call the new model $amf2$

We make three graphs
• the first sets the stage by repeating an earlier graph
• the second contains two histograms of values of log likelihoods of the two models over the period $T$
• the third compares likelihoods under the true and alternative models

Heres the code

```python
# Create the second (wrong) alternative model
A2, B2, D2, F2 = [0.9, 1.0, 0.55, 0.25]  # parameters for $\theta_1$ closer to $\Theta_0$
amf2 = AMF_LSS_VAR(A2, B2, D2, F2)

# Get likelihood from each path $x^{(i)}, y^{(i)}$
LLit2 = simulate_likelihood(amf2, Xit, Yit)

LLT2 = 1/(T-1) * LLit2[:, -1]
LLmean_t2 = np.mean(LLT2)

fig, ax = plt.subplots()
ax.hist(LLT2)
ax.vlines(LLmean_t2, ymin=0, ymax=1400, color="k", linestyle="--", alpha=0.6)
plt.title(r"Distribution of $\frac{1}{T}\log L_T | \theta_1$"
plt.show()
```

The resulting figure looks like this

![Distribution of $\frac{1}{T}\log L_T | \theta_1$](image)

Lets see a histogram of the log-likelihoods under the true and the alternative model (same sample paths)
Now well plot the histogram of the difference in log likelihood ratio

```python
LLT_diff = LLT - LLT2
fig, ax = plt.subplots(figsize=(8, 6))
ax.hist(LLT_diff, bins=50)
plt.title(r"""$rac{1}{T} \left[ \log (L_T^i \mid \theta_0) - \log (L_T^i \mid \theta_1) \right]$""
plt.show()
```
The resulting figure is as follows

\[
\frac{1}{T} \left[ \log(L_T^i \mid \theta_0) - \log(L_T^i \mid \theta_1) \right]
\]

**Interpretation**

These histograms of log likelihood ratios illustrate important features of **likelihood ratio tests** as tools for discriminating between statistical models

- The log likelihood is higher on average under the true model – obviously a very useful property
- Nevertheless, for a positive fraction of realizations, the log likelihood is higher for the incorrect than for the true model
- In these instances, a likelihood ratio test mistakenly selects the wrong model
- These mechanics underlie the statistical theory of **mistake probabilities** associated with model selection tests based on likelihood ratio

(In a subsequent lecture, well use some of the code prepared in this lecture to illustrate mistake probabilities)
8.4.3 Benefits from Reduced Aggregate Fluctuations

Now lets turn to a new example of multiplicative functionals

This example illustrates ideas in the literatures on

- **long-run risk** in the consumption based asset pricing literature (e.g., [BY04], [HHL08], [Han07])
- **benefits of eliminating aggregate fluctuations** in representative agent macro models (e.g., [Tal00], [Luc03])

Let $c_t$ be consumption at date $t \geq 0$.

Suppose that $\{\log c_t\}_{t=0}^{\infty}$ is an additive functional described by

$$
\log c_{t+1} - \log c_t = \nu + D \cdot x_t + F \cdot z_{t+1}
$$

where

$$
x_{t+1} = Ax_t + Bz_{t+1}
$$

Here $\{z_{t+1}\}_{t=0}^{\infty}$ is an i.i.d. sequence of $\mathcal{N}(0, I)$ random vectors.

A representative household ranks consumption processes $\{c_t\}_{t=0}^{\infty}$ with a utility functional $\{V_t\}_{t=0}^{\infty}$ that satisfies

$$
\log V_t - \log c_t = U \cdot x_t + u
$$

where

$$
U = \exp(-\delta) \left[ I - \exp(-\delta) A \right]^{-1} D
$$

and

$$
u = \frac{\exp(-\delta)}{1 - \exp(-\delta)} \nu + \frac{1 - \gamma}{2} \frac{\exp(-\delta)}{1 - \exp(-\delta)} \left| D' \left[ I - \exp(-\delta) A \right]^{-1} B + F \right|^2,
$$

Here $\gamma \geq 1$ is a risk-aversion coefficient and $\delta > 0$ is a rate of time preference.

**Consumption as a multiplicative process**

We begin by showing that consumption is a **multiplicative functional** with representation

$$
c_t / c_0 = \exp(\tilde{\nu} t) \begin{pmatrix} \tilde{M}_t \\ \tilde{M}_0 \end{pmatrix} \begin{pmatrix} \tilde{e}(x_0) \\ \tilde{e}(x_t) \end{pmatrix}
$$

where $\left( \tilde{M}_t / \tilde{M}_0 \right)$ is a likelihood ratio process and $\tilde{M}_0 = 1$. 

8.4. Multiplicative Functionals
At this point, as an exercise, we ask the reader please to verify the following formulas for $\bar{\nu}$ and $\bar{e}(x_t)$ as functions of $A, B, D, F$:

$$\bar{\nu} = \nu + \frac{H \cdot H}{2}$$

and

$$\bar{e}(x) = \exp[g(x)] = \exp[D'(I - A)^{-1}x]$$

**Simulating a likelihood ratio process again**

Next, we want a program to simulate the likelihood ratio process $\bar{M}_t$ again.

In particular, we want to simulate 5000 sample paths of length $T = 1000$ 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 a histogram of $\bar{M}_T$ for $T = 1000$.

Here is code that accomplishes these tasks:

```python
def simulate_martingale_components(amf, T=1000, I=5000):
    # Get the multiplicative decomposition
    \nu, 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, 0.005)

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))}")
```
Heres the output:

| The (min, mean, max) of additive Martingale component in period T is | (-1.7419029969162607, -0.009316975586058086, 2.091259035641934) |
| The (min, mean, max) of multiplicative Martingale component in period T is | (0.15656398590834272, 0.9919363162991409, 7.234574417683094) |

Comments

- The preceding min, mean, and max of the cross-section of the date $T$ realizations of the multiplicative martingale component of $c_t$ indicate that the sample mean is close to its population mean of 1
  - This outcome prevails for all values of the horizon $T$
- The cross-section distribution of the multiplicative martingale component of $c$ at date $T$ approximates a log normal distribution well
- The histogram of the additive martingale component of $\log c_t$ at date $T$ approximates a normal distribution well

Heres a histogram of the additive martingale component

```python
fig, ax = plt.subplots(figsize=(8, 6))
ax.hist(amcT, bins=25, normed=True)
plt.title("Histogram of Additive Martingale Component")
plt.show()
```
Here's a histogram of the multiplicative martingale component:

```python
fig, ax = plt.subplots(figsize=(8, 6))
ax.hist(mmcT, bins=25, normed=True)
plt.title("Histogram of Multiplicative Martingale Component")
plt.show()
```
Representing the likelihood ratio process

The likelihood ratio process \( \{\tilde{M}_t\}_{t=0}^\infty \) can be represented as

\[
\tilde{M}_t = \exp \left( \sum_{j=1}^{t} \left( H \cdot z_j - \frac{H \cdot H}{2} \right) \right), \quad \tilde{M}_0 = 1,
\]

where \( H = [F + B'(I - A')^{-1}D] \)

It follows that \( \log \tilde{M}_t \sim \mathcal{N}(-\frac{tH \cdot H}{2}, tH \cdot H) \) and that consequently \( \tilde{M}_t \) is log normal.

Let's plot the probability density functions for \( \log \tilde{M}_t \) 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 \( \log \tilde{M}_t \) for different values of \( t \).

Note: \texttt{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 \texttt{scale} (scale=exp(\(-tH \cdot H/2\)))
• 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

```python
def Mtilde_t_density(amf, t, xmin=1e-8, xmax=5.0, npts=5000):
    # Pull out the multiplicative decomposition
    tilde, 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
    tilde, 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()
```

Here's the output:
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

8.4. Multiplicative Functionals
These probability density functions illustrate a peculiar property of log likelihood ratio processes:

- With respect to the true model probabilities, they have mathematical expectations equal to 1 for all $t \geq 0$
- They almost surely converge to zero

**Welfare benefits of reduced random aggregate fluctuations**

Suppose in the tradition of a strand of macroeconomics (for example Tallarini [Tal00], [Luc03]) we want to estimate the welfare benefits from removing random fluctuations around trend growth.

We shall compute how much initial consumption $c_0$ a representative consumer who ranks consumption streams according to (8.25) would be willing to sacrifice to enjoy the consumption stream

$$\frac{c_t}{c_0} = \exp(\tilde{\nu} t)$$

rather than the stream described by equation (8.26).

We want to compute the implied percentage reduction in $c_0$ that the representative consumer would accept.

To accomplish this, we write a function that computes the coefficients $U$ and $u$ for the original values of $A, B, D, F, \nu$, but also for the case that $A, B, D, F = [0, 0, 0, 0]$ and $\nu = \tilde{\nu}$.

Here is our code:

```python
def Uu(amf, \delta, \gamma):
    \nu_tilde, H, g = amf.multiplicative_decomp()
    resolv = 1 / (1 - np.exp(-\delta) * A)
    vect = F + D * resolv * B
    U_risky = np.exp(-\delta) * resolv * D
    u_risky = (np.exp(-\delta) / (1 - np.exp(-\delta))) * (\nu + (.5) * (1 - \gamma) * \nu_tilde)

    U_det = 0
    u_det = (np.exp(-\delta) / (1 - np.exp(-\delta))) * \nu_tilde

    return U_risky, u_risky, U_det, u_det
```

# Set remaining parameters
\delta = 0.02
\gamma = 2.0

# Get coeffs
U_r, u_r, U_d, u_d = Uu(amf_2, \delta, \gamma)

The values of the two processes are

$$\log V^r_0 = \log c^r_0 + U^r x_0 + u^r$$
$$\log V^d_0 = \log c^d_0 + U^d x_0 + u^d$$
We look for the ratio $\frac{c^d}{c^r}$ that makes $\log V^r_0 - \log V^d_0 = 0$

$$\log V^r_0 - \log V^d_0 + \log c^d_0 - \log c^r_0 = (U^r - U^d)x_0 + u^r - u^d$$

$$\frac{c^d_0}{c^r_0} = \exp \left( (U^r - U^d)x_0 + u^r - u^d \right)$$

Hence, the implied percentage reduction in $c_0$ that the representative consumer would accept is given by

$$\frac{c^r_0 - c^d_0}{c^r_0} = 1 - \exp \left( (U^r - U^d)x_0 + u^r - u^d \right)$$

Let's compute this

```python
x0 = 0.0  # initial conditions
logVC_r = U_r * x0 + u_r
logVC_d = U_d * x0 + u_d
perc_reduct = 100 * (1 - np.exp(logVC_r - logVC_d))
perc_reduct
```

1.0809878812017448

We find that the consumer would be willing to take a percentage reduction of initial consumption equal to around 1.081

**Contents**

- Classical Control with Linear Algebra
  - Overview
  - A Control Problem
  - Finite Horizon Theory
  - The Infinite Horizon Limit
  - Undiscounted Problems
  - Implementation
  - Exercises

**8.5 Classical Control with Linear Algebra**

**8.5.1 Overview**

In an earlier lecture *Linear Quadratic Dynamic Programming Problems* we have studied how to solve a special class of dynamic optimization and prediction problems by applying the method of dynamic programming. In this class of problems
• the objective function is **quadratic** in states and controls
• the one-step transition function is **linear**
• shocks are i.i.d. 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 well put to work here is **LU decomposition**.

Well begin with discrete horizon problems.

Then well 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.

**References**

Useful references include [Whi63], [HS80], [Orf88], [AP91], and [Mut60].

### 8.5.2 A Control Problem

Let $L$ be the **lag operator**, so that, for sequence $\{x_t\}$ we have $Lx_t = x_{t-1}$.

More generally, let $L^k x_t = x_{t-k}$ with $L^0 x_t = x_t$ and

$$d(L) = d_0 + d_1 L + \ldots + d_m L^m$$

where $d_0, d_1, \ldots, d_m$ is a given scalar sequence.

Consider the discrete time control problem

$$\max_{\{y_t\}} \lim_{N \to \infty} \sum_{t=0}^{N} \beta^t \left\{ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} [d(L)y_t]^2 \right\}, \quad (8.27)$$

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 $\lim_{t \to \infty} \beta^{t/2} a_t = 0$

Maximization in (8.27) is subject to initial conditions for $y_{-1}, y_{-2}, \ldots, y_{-m}$.
Maximization is over infinite sequences $\{y_t\}_{t \geq 0}$.

**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 Dynamic Programming Problems* 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

$$\pi_t = p_t q_t - c q_t - \gamma (q_{t+1} - q_t)^2$$

with $p_t = \alpha_0 - \alpha_1 q_t + d_t$.

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 (8.27) when

• $a_t := \alpha_0 + d_t - c$

• $h := 2\alpha_1$

• $d(L) := \sqrt{2\gamma} (I - L)$

Further examples of this problem for factor demand, economic growth, and government policy problems are given in ch. IX of [Sar87].

### 8.5.3 Finite Horizon Theory

We first study a finite $N$ version of the problem.

Later we will study an infinite horizon problem solution as a limiting version of a finite horizon problem.

(This will require being careful because the limits as $N \to \infty$ of the necessary and sufficient conditions for maximizing finite $N$ versions of (8.27) are not sufficient for maximizing (8.27)).

We begin by

1. fixing $N > m$,

2. differentiating the finite version of (8.27) with respect to $y_0, y_1, \ldots, y_N$, and

3. setting these derivatives to zero.
For \( t = 0, \ldots, N - m \) these first-order necessary conditions are the *Euler equations*

For \( t = N - m + 1, \ldots, N \), the first-order conditions are a set of *terminal conditions*

Consider the term

\[
J = \sum_{t=0}^{N} \beta^t [d(L)y_t][d(L)y_t]
\]

\[
= \sum_{t=0}^{N} \beta^t \left( d_0 y_t + d_1 y_{t-1} + \cdots + d_m y_{t-m} \right) \left( d_0 y_t + d_1 y_{t-1} + \cdots + d_m y_{t-m} \right)
\]

Differentiating \( J \) with respect to \( y_t \) for \( t = 0, 1, \ldots, N - m \) gives

\[
\frac{\partial J}{\partial y_t} = 2\beta^t d_0 d(L)y_t + 2\beta^{t+1} d_1 d(L)y_{t+1} + \cdots + 2\beta^{t+m} d_m d(L)y_{t+m}
\]

\[
= 2\beta^t \left( d_0 + d_1 \beta L^{-1} + d_2 \beta^2 L^{-2} + \cdots + d_m \beta^m L^{-m} \right) d(L)y_t
\]

We can write this more succinctly as

\[
\frac{\partial J}{\partial y_t} = 2\beta^t d(\beta L^{-1}) d(L)y_t
\]

Differentiating \( J \) with respect to \( y_t \) for \( t = N - m + 1, \ldots, N \) gives

\[
\frac{\partial J}{\partial y_{N-m}} = 2\beta^{N-m+1} \left[ d_0 + \beta L^{-1} d_1 + \cdots + \beta^{m-1} L^{-m+1} d_m \right] d(L)y_{N-m+1}
\]

With these preliminaries under our belts, we are ready to differentiate (8.27)

Differentiating (8.27) with respect to \( y_t \) for \( t = 0, \ldots, N - m \) gives the Euler equations

\[
[h + d(\beta L^{-1}) d(L)]y_t = a_t, \quad t = 0, 1, \ldots, N - m
\]

The system of equations (8.30) form a \( 2 \times m \) order linear *difference equation* that must hold for the values of \( t \) indicated.

Differentiating (8.27) with respect to \( y_t \) for \( t = N - m + 1, \ldots, N \) gives the terminal conditions
\[
\beta^N(a_N - hy_N - d_0 d(L)y_N) = 0 \\
\beta^{N-1}(a_{N-1} - hy_{N-1} - (d_0 + \beta d_1 L^{-1}) d(L)y_{N-1}) = 0 \\
\vdots \\
\beta^{N-m+1}(a_{N-m+1} - hy_{N-m+1} - (d_0 + \beta L^{-1}d_1 + \cdots + \beta^{m-1}L^{-m+1}d_{m-1})d(L)y_{N-m+1}) = 0
\] (8.31)

In the finite \(N\) problem, we want simultaneously to solve (8.30) subject to the \(m\) initial conditions \(y_1, \ldots, y_m\) and the \(m\) terminal conditions (8.31).

These conditions uniquely pin down the solution of the finite \(N\) problem. That is, for the finite \(N\) problem, conditions (8.30) and (8.31) are necessary and sufficient for a maximum, by concavity of the objective function.

Next we describe how to obtain the solution using matrix methods.

**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

\[
\begin{bmatrix}
[ h + d(\beta L^{-1}) d(L) ] y_t = a_t, & t = 0, 1, \ldots, N - 1 \\
\beta^N \left[ a_N - h y_N - d_0 d(L)y_N \right] = 0 
\end{bmatrix}
\] (8.32)

where \(d(L) = d_0 + d_1 L\).

These equations are to be solved for \(y_0, y_1, \ldots, y_N\) as functions of \(a_0, a_1, \ldots, a_N\) and \(y_{-1}\).

Let

\[
\phi(L) = \phi_0 + \phi_1 L + \beta \phi_1 L^{-1} = h + d(\beta L^{-1})d(L) = (h + d_0^2 + d_1^2) + d_1 d_0 L + d_1 d_0 \beta L^{-1}
\]

Then we can represent (8.32) as the matrix equation

\[
\begin{bmatrix}
(\phi_0 - d_1^2) & \phi_1 & 0 & 0 & \ldots & \ldots & 0 \\
\beta \phi_1 & \phi_0 & \phi_1 & 0 & \ldots & \ldots & 0 \\
0 & \beta \phi_1 & \phi_0 & \phi_1 & \ldots & \ldots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & \ldots & \ldots & \ldots & \beta \phi_1 & \phi_0 & \phi_1 \\
0 & \ldots & \ldots & \ldots & 0 & \beta \phi_1 & \phi_0 
\end{bmatrix}
\begin{bmatrix}
y_N \\
y_{N-1} \\
y_{N-2} \\
\vdots \\
y_1 \\
y_0 
\end{bmatrix}
= 
\begin{bmatrix}
a_N \\
a_{N-1} \\
a_{N-2} \\
\vdots \\
a_1 \\
a_0 - \phi_1 y_{-1} 
\end{bmatrix}
\] (8.33)
or

\[ W \tilde{y} = \tilde{a} \quad (8.34) \]

Notice how we have chosen to arrange the \( y_t \)'s in reverse time order.

The matrix \( W \) on the left side of (8.33) 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 subdiagonal elements equal \( \beta \) time the superdiagonal elements.

The solution of (8.34) can be expressed in the form

\[ \tilde{y} = W^{-1} \tilde{a} \quad (8.35) \]

which represents each element \( y_t \) of \( \tilde{y} \) as a function of the entire vector \( \tilde{a} \).

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 (8.33) or (8.34) 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 (8.35), we obtain

\[ U \tilde{y} = L^{-1} \tilde{a} \quad (8.36) \]

Since \( L^{-1} \) is lower triangular, this representation expresses \( y_t \) as a function of

- lagged \( y \)s (via the term \( U \tilde{y} \)), and
- current and future \( a \)s (via the term \( L^{-1} \tilde{a} \)).

Because there are zeros everywhere in the matrix on the left of (8.33) except on the diagonal, superdiagonal, and subdiagonal, the \( LU \) decomposition takes...
• $L$ to be zero except in the diagonal and the leading subdiagonal
• $U$ to be zero except on the diagonal and the superdiagonal

Thus, (8.36) has the form

\[
\begin{bmatrix}
1 & U_{12} & 0 & 0 & \ldots & 0 & 0 \\
0 & 1 & U_{23} & 0 & \ldots & 0 & 0 \\
0 & 0 & 1 & U_{34} & \ldots & 0 & 0 \\
0 & 0 & 0 & 1 & \ldots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & 0 & \ldots & 1 & U_{N,N+1} \\
0 & 0 & 0 & 0 & \ldots & 0 & 1
\end{bmatrix}
\begin{bmatrix}
y_N \\
y_{N-1} \\
y_{N-2} \\
y_{N-3} \\
\vdots \\
y_1 \\
y_0
\end{bmatrix}
= 
\begin{bmatrix}
L_{11}^{-1} & 0 & 0 & \ldots & 0 \\
L_{21}^{-1} & L_{22}^{-1} & 0 & \ldots & 0 \\
L_{31}^{-1} & L_{32}^{-1} & L_{33}^{-1} & \ldots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
L_{N,1}^{-1} & L_{N,2}^{-1} & L_{N,3}^{-1} & \ldots & 0 \\
L_{N+1,1}^{-1} & L_{N+1,2}^{-1} & L_{N+1,3}^{-1} & \ldots & L_{N+1,N+1}^{-1}
\end{bmatrix}
\begin{bmatrix}
a_N \\
a_{N-1} \\
a_{N-2} \\
a_1 \\
a_0 - \phi_1 y_{-1}
\end{bmatrix}
\]

where $L_{ij}^{-1}$ is the $(i,j)$ element of $L^{-1}$ and $U_{ij}$ is the $(i,j)$ element of $U$

Note how the left side for a given $t$ involves $y_t$ and one lagged value $y_{t-1}$ while the right side involves all future values of the forcing process $a_t, a_{t+1}, \ldots, a_N$

### Additional Lag Terms

We briefly indicate how this approach extends to the problem with $m > 1$

Assume that $\beta = 1$ and let $D_{m+1}$ be the $(m+1) \times (m+1)$ symmetric matrix whose elements are determined from the following formula:

\[
D_{jk} = d_0 d_{k-j} + d_1 d_{k-j+1} + \ldots + d_{j-1} d_{k-j}, \quad k \geq j
\]

Let $I_{m+1}$ be the $(m+1) \times (m+1)$ identity matrix

Let $\phi_j$ be the coefficients in the expansion $\phi(L) = h + d(L^{-1})d(L)$

Then the first order conditions (8.30) and (8.31) 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

\[
M_{ij} = \begin{cases} 
D_{i-j,m+1} & \text{for } i > j \\
0 & \text{for } i \leq j
\end{cases}
\]

8.5. Classical Control with Linear Algebra
\[ \phi_m y_{N-1} + \phi_{m-1} y_{N-2} + \ldots + \phi_0 y_{N-m-1} + \phi_1 y_{N-m-2} + \ldots + \phi_m y_{N-2m-1} = a_{N-m-1} \]
\[ \phi_m y_{N-2} + \phi_{m-1} y_{N-3} + \ldots + \phi_0 y_{N-m-2} + \phi_1 y_{N-m-3} + \ldots + \phi_m y_{N-2m-2} = a_{N-m-2} \]
\[ \vdots \]
\[ \phi_m y_{m+1} + \phi_{m-1} y_{m} + \ldots + \phi_0 y_1 + \phi_1 y_0 + \phi_m y_{m+1} = a_1 \]
\[ \phi_m y_m + \phi_{m-1} y_{m-1} + \phi_{m-2} y_{m-2} + \ldots + \phi_0 y_0 + \phi_1 y_{-1} + \ldots + \phi_m y_{m} = a_0 \]

As before, we can express this equation as \( W \tilde{y} = \tilde{a} \)

The matrix on the left of this equation is almost Toeplitz, the exception being the leading \( m \times m \) sub matrix in the upper left hand corner.

We can represent the solution in feedback-feedforward form by obtaining a decomposition \( LU = W \), and obtain

\[ U \tilde{y} = L^{-1} \tilde{a} \quad (8.37) \]

\[ \sum_{j=0}^{t} U_{-t+N+1,-t+N+j+1} y_{t-j} = \sum_{j=0}^{N-t} L_{-t+N+1,-t+N+1-j} \tilde{a}_{t+j} \]
\[ t = 0, 1, \ldots, N \]

where \( L_{t,s}^{-1} \) is the element in the \((t, s)\) position of \( L \), and similarly for \( U \)

The left side of equation \((8.37)\) 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 \( N \to \infty \), \((8.37)\) 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 \( t \), \( U_{t, t-j}^{-1} \to c_j \) 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.

### 8.5.4 The Infinite Horizon Limit

For the infinite horizon problem, we propose to discover first-order necessary conditions by taking the limits of \((8.30)\) and \((8.31)\) as \( N \to \infty \).
This approach is valid, and the limits of (8.30) and (8.31) as \( N \) approaches infinity are first-order necessary conditions for a maximum.

However, for the infinite horizon problem with \( \beta < 1 \), the limits of (8.30) and (8.31) are, in general, not sufficient for a maximum.

That is, the limits of (8.31) do not provide enough information uniquely to determine the solution of the Euler equation (8.30) that maximizes (8.27).

As we shall see below, a side condition on the path of \( y_t \) that together with (8.30) is sufficient for an optimum is

\[
\sum_{t=0}^{\infty} \beta^t h y_t^2 < \infty
\]  

(8.38)

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 (8.27) than does the optimal path selected by the solution procedure below.

Consider the characteristic equation associated with the Euler equation

\[
h + d(\beta z^{-1}) d(z) = 0
\]  

(8.39)

Notice that if \( \tilde{z} \) is a root of equation (8.39), then so is \( \beta \tilde{z}^{-1} \).

Thus, the roots of (8.39) come in \( \beta \)-reciprocal pairs.

Assume that the roots of (8.39) 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 (8.39) can be expressed as

\[
h + d(\beta z^{-1}) d(z) = z^{-m} z_0 (z - z_1) \cdots (z - z_m) (z - z_{m+1}) \cdots (z - z_{2m})
\]

\[
= z^{-m} z_0 (z - z_1) (z - z_2) \cdots (z - z_m) (z - \beta z_{m-1}) \cdots (z - \beta z_2^{-1}) (z - \beta z_1^{-1})
\]  

(8.40)

where \( z_0 \) is a constant.

In (8.40), we substitute \( (z - z_j) = -z_j (1 - \frac{1}{z_j} z) \) and \( (z - \beta z_j^{-1}) = z (1 - \frac{\beta}{z_j} z^{-1}) \) for \( j = 1, \ldots, m \) to get

\[
h + d(\beta z^{-1}) d(z) = (-1)^m (z_0 z_1 \cdots z_m) (1 - \frac{1}{z_1} z) \cdots (1 - \frac{1}{z_m} z) (1 - \frac{1}{z_1} \beta z^{-1}) \cdots (1 - \frac{1}{z_m} \beta z^{-1})
\]

Now define \( c(z) = \sum_{j=0}^{m} c_j z^j \) as
\[ c(z) = \left[ (-1)^m z_0 z_1 \cdots z_m \right]^{1/2} \left( 1 - \frac{z}{z_1} \right) \left( 1 - \frac{z}{z_2} \right) \cdots \left( 1 - \frac{z}{z_m} \right) \] (8.41)

Notice that (8.40) can be written

\[ h + d(\beta z^{-1}) d(z) = c(\beta z^{-1}) c(z) \] (8.42)

It is useful to write (8.41) as

\[ c(z) = c_0 (1 - \lambda_1 z) \cdots (1 - \lambda_m z) \] (8.43)

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 (8.43), we can express the factorization (8.42) as

\[ h + d(\beta z^{-1}) d(z) = c_0^2 (1 - \lambda_1 z) \cdots (1 - \lambda_m z)(1 - \lambda_1 z^{-1}) \cdots (1 - \lambda_m z^{-1}) \]

In sum, we have constructed a factorization (8.42) 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 (8.42), 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 (8.38) is

\[ c(L) y_t = c(\beta L^{-1})^{-1} a_t \] (8.44)

This can be established by using an argument paralleling that in chapter IX of [Sar87]

To exhibit the solution in a form paralleling that of [Sar87], we use (8.43) to write (8.44) as

\[ (1 - \lambda_1 L) \cdots (1 - \lambda_m L) y_t = \frac{c_0^{-2} a_t}{(1 - \beta \lambda_1 L^{-1}) \cdots (1 - \beta \lambda_m L^{-1})} \] (8.45)

Using partial fractions, we can write the characteristic polynomial on the right side of (8.45) 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 (8.45) 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

\[(1 - \lambda_1 L) \cdots (1 - \lambda_m L)y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} (\lambda_j \beta)^k a_{t+k}\]  \hspace{1cm} (8.46)

Equation (8.46) expresses the optimum sequence for \(y_t\) in terms of \(m\) lagged \(y\)s, and \(m\) weighted infinite geometric sums of future \(a\)s.

Furthermore, (8.46) is the unique solution of the Euler equation that satisfies the initial conditions and condition (8.38).

In effect, condition (8.38) compels us to solve the unstable roots of \(h + d(\beta z^{-1})d(z)\) forward (see [Sar87]).

The step of factoring the polynomial \(h + d(\beta z^{-1})d(z)\) into \(c(\beta z^{-1})c(z)\), where the zeros of \(c(z)\) all have modulus exceeding \(\sqrt{\beta}\), is central to solving the problem.

We note two features of the solution (8.46):

- 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 \(a\)s on the right side of (8.46) converge.

We immediately see that those sums will converge under the weaker condition that \(\{a_t\}\) is of exponential order less than \(\phi^{-1}\) where \(\phi = \max\{\beta \lambda_i, i = 1, \ldots, m\}\).

Note that with \(a_t\) identically zero, (8.46) 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 (8.38) is satisfied.

In fact, \(\max_i |\lambda_i| < 1/\sqrt{\beta}\) is a necessary condition for (8.38) to hold.

Were (8.38) 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 (8.43).

We can simply let \(y_t = 0\) for \(t \geq 0\).

This policy involves at most \(m\) nonzero values of \(h y_t^2\) and \([d(L)y_t]^2\), and so yields a finite value of (8.27).

Therefore it is easy to dominate a path that violates (8.38).

### 8.5.5 Undiscounted Problems

It is worthwhile focusing on a special case of the LQ problems above: the undiscounted problem that emerges when \(\beta = 1\).
In this case, the Euler equation is
\[
\left(h + d(L^{-1})d(L)\right) y_t = a_t
\]
The factorization of the characteristic polynomial (8.42) becomes
\[
\left(h + d(z^{-1})d(z)\right) = c(z^{-1}) c(z)
\]
where
\[
c(z) = c_0(1 - \lambda_1 z) \cdots (1 - \lambda_m z) \\
c_0 = \left(-1\right)^m z_0 z_1 \cdots z_m \\
|\lambda_j| < 1 \text{ for } j = 1, \ldots, m \\
\lambda_j = \frac{1}{z_j} \text{ for } j = 1, \ldots, m \\
z_0 = \text{ constant}
\]
The solution of the problem becomes
\[
(1 - \lambda_1 L) \cdots (1 - \lambda_m L) y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} \lambda_j^k a_{t+k}
\]

**Transforming discounted to undiscounted problem**

Discounted problems can always be converted into undiscounted problems via a simple transformation

Consider problem (8.27) with \(0 < \beta < 1\)

Define the transformed variables
\[
\tilde{a}_t = \beta^{t/2} a_t, \quad \tilde{y}_t = \beta^{t/2} y_t
\]

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 (8.27) is equivalent to

\[
\lim_{N \to \infty} \sum_{t=0}^{N} \left\{ \tilde{a}_t \tilde{y}_t - \frac{1}{2} h \tilde{y}_t^2 - \frac{1}{2} [\tilde{d}(L)\tilde{y}_t]^2 \right\}
\]

which is to be maximized over sequences \(\{\tilde{y}_t, \ t = 0, \ldots\}\) subject to \(\tilde{y}_{-1}, \ldots, \tilde{y}_{-m}\) given and \(\{\tilde{a}_t, \ t = 1, \ldots\}\) a known bounded sequence

The Euler equation for this problem is \([h + \tilde{d}(L^{-1}) \tilde{d}(L)] \tilde{y}_t = \tilde{a}_t\)

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

\[
\tilde{y}_t = \tilde{f}_1 \tilde{y}_{t-1} + \cdots + \tilde{f}_m \tilde{y}_{t-m} + \sum_{j=1}^m \tilde{A}_j \sum_{k=0}^\infty \tilde{\lambda}_j^k \tilde{a}_{t+k}, \tag{8.49}
\]

where \( \tilde{c}(z^{-1})\tilde{c}(z) = h + \tilde{d}(z^{-1})\tilde{d}(z) \), and where

\[
[(1)^m \tilde{z}_0 \tilde{z}_1 \cdots \tilde{z}_m]^{1/2} (1 - \tilde{\lambda}_1 z) \cdots (1 - \tilde{\lambda}_m z) = \tilde{c}(z), \text{ where } |\tilde{\lambda}_j| < 1
\]

We leave it to the reader to show that (8.49) 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

\[
f_j = \tilde{f}_j \beta^{-j/2}, \quad A_j = \tilde{A}_j, \quad \lambda_j = \tilde{\lambda}_j \beta^{-1/2}
\tag{8.50}
\]

The transformations (8.47) and the inverse formulas (8.50) allow us to solve a discounted problem by first solving a related undiscounted problem.

### 8.5.6 Implementation

Code that computes solutions to the LQ problem using the methods described above can be found in file `control_and_filter.py`

Here’s how it looks:

```python
""
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, \beta=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
""
```

8.5. Classical Control with Linear Algebra 1063
Parameter of the objective function (corresponding to the quadratic term)

\[ y_m : \text{list or numpy.array (1-D or a 2-D column vector)} \]

Initial conditions for \( y \)

\[ r : \text{list or numpy.array (1-D or a 2-D column vector)} \]

The order of the coefficients: \([r_0, r_1, \ldots, r_k]\) (optional, if not defined \(\rightarrow\) deterministic problem)

\[ \beta : \text{scalar} \]

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 up front
#---------------------------------------------
# = np.zeros(2 * self.m + 1)
for i in range(-self.m, self.m + 1):
    [self.m - i] = np.sum(np.diag(self.d.reshape(self.m + 1, 1) @
                      self.d.reshape(1, self.m + 1)), k=-i))
    [self.m] = [self.m] + self.h
self. =

#---------------------------------------------
# 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
    _r = np.zeros(2 * self.k + 1)
    for i in range(-self.k, self.k + 1):
        _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._r = _r
else:
    _r[self.k] = _r[self.k] + h_eps
    self._r = _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) Construct 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)
    #----------------------------------------------

    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)):
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)
    """
    m = self.m
    # Calculate the roots of the 2m-polynomial
    roots = np.roots()
    # sort the roots according to their length (in descending order)
    roots_sorted = roots[np.argsort(abs(roots))[-1:]]
    z_0 = np.sum() / np.poly1d(roots, True)(1)
    z_1_to_m = roots_sorted[:m]  # we need only those outside the unit circle
    lambda_ = 1 / z_1_to_m
    return z_1_to_m, z_0, lambda_

def coeffs_of_c(self):
    """
    This function computes the coefficients \{c_j, j = 0, 1, \ldots, m\} for
    \[ c(z) = \sum_{j = 0}^{m} c_j z^j \]
    Based on the expression (1.9). The order is
    \[ c_{\text{coeffs}} = \{c_0, c_1, \ldots, c_{(m-1)}, c_m\} \]
    \[ z_1_{\text{to}_m}, z_0 = self.roots_of_characteristic()[:2] \]
    \[ c_0 = (z_0 + \text{np}.prod(z_1_{\text{to}_m}).\text{real} + (-1)^{\text{np}.\text{imag}(z_1_{\text{to}_m})})^{0.5} \]
    \[ c_{\text{coeffs}} = \text{np}.\text{poly1d}(z_1_{\text{to}_m}, True).c \times z_0 / c_0 \]
    return c_coeffs[:-1]

W[row, (i + 1):(2 * m + 2 + i)] =

for i in range(1, m + 1):
    W[N - m + i, -(2 * m + 1 - i)] = [-i]

for i in range(m):
    W_m[N - i, :(m - i)] = [(m + 1 + i):]

return W, W_m
```python
def solution(self):
    """
    This function calculates \{\lambda_j, j=1,...,m\} and \{A_j, j=1,...,m\}
    of the expression (1.15)
    """
    \lambda = 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 - \lambda[j]
        A[j] = c_0 ** (-2) / np.prod(denom[np.arange(self.m) != j])

    return \lambda, 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))
    _r = self._r

    for i in range(N):
        for j in range(N):
            if abs(i-j) <= self.k:
                V[i, j] = _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)
```

8.5. Classical Control with Linear Algebra 1067
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 determinstic 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 cosistent 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 = J @ np.vstack([y_bar, self.y_m])  # y_hist : concatenated y_m and y_bar

#--------------------------------------------
# Transform the optimal sequence back if \beta is given
#--------------------------------------------
if self.\beta != 1:
y_hist = y_hist * (self.\beta**(-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 = J @ np.vstack([y_bar, self.y_m])  # y_hist : concatenated y_m and y_bar

return y_hist, L, U, y_bar

**Example**

In this application we 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 well set \( h = 1 \), so that the agent just wants to track the \( \{a_t\} \) process

However, as we increase \( \gamma \), the agent gives greater weight to a smooth time path

---

8.5. Classical Control with Linear Algebra 1069
Hence \( \{ y_t \} \) evolves as a smoothed version of \( \{ a_t \} \).

The \( \{ a_t \} \) sequence well choose as a stationary cyclic process plus some white noise.

Here’s some code that generates a plot when \( \gamma = 0.8 \):

```python
import matplotlib.pyplot as plt

# == 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(\gamma=.8, \text{m}=1, \text{h}=1, \text{y}_\text{m}=2):
    d = \gamma * np.asarray([1, -1])
    y_m = np.asarray(y_m).reshape(\text{m}, 1)
    testlq = LQFilter(d, \text{h}, y_m)
    y_hist, L, U, y = testlq.optimal_y(a_seq)
    y = y[::\text{-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 / \text{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 $\gamma$={\gamma}', xlabel='Time', xlim=(0, max(time)))
    ax.legend()
    ax.grid()
    plt.show()

plot_simulation()
```
Here's what happens when we change $\gamma$ to 5.0

```python
plot_simulation(\gamma=5)
```
And heres $\gamma = 10$

```python
plot_simulation(\gamma=10)
```
8.5.7 Exercises

Exercise 1

Consider solving a discounted version ($\beta < 1$) of problem (8.27), as follows

Convert (8.27) to the undiscounted problem (8.48)

Let the solution of (8.48) 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

$$\tilde{y}_t = \tilde{f}_1 \tilde{y}_{t-1} + \cdots + \tilde{f}_m \tilde{y}_{t-m} + \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k} \quad (8.51)$$

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 (8.51) 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 \( \lambda_j = \tilde{\lambda}_j \beta^{-j/2} \)

**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 (8.46), giving numerical values for the coefficients

Make sure that the boundary conditions (8.31) are satisfied

(Note: this problem differs from the problem in the text in one important way: instead of \( h > 0 \) in (8.27), \( h = 0 \). This has an important influence on the solution.)

**Exercise 3**

Solve the infinite time optimal control problem to maximize

\[
\lim_{N \to \infty} \sum_{t=0}^N \left\{ a_t y_t - \frac{1}{2} [(1 - 2L) y_t]^2 \right\},
\]

subject to \( y_{-1} \) given. Prove that the solution is

\[
y_t = 2y_{t-1} = 2^{t+1} y_{-1}, \quad t > 0
\]

**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 (8.38), and so is not optimal

Prove that the optimal solution is approximately \( y_t = .5y_{t-1} \)
8.6 Classical Filtering With Linear Algebra

8.6.1 Overview

This is a sequel to the earlier lecture *Classical Control with Linear Algebra*. That lecture used linear algebra – in particular, the LU decomposition – to formulate and solve a class of linear-quadratic optimal control problems.

In this lecture, we will 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 type of problem.

This connection, which is often termed duality, is present whether one uses classical or recursive solution procedures.

In fact, we saw duality at work earlier when we formulated control and prediction problems recursively in lectures *LQ dynamic programming problems, A first look at the Kalman filter*, and *The permanent income model*.

A useful consequence of duality is that:

- With every LQ control problem there is implicitly affiliated a linear least squares prediction or filtering problem.
- With every linear least squares prediction or filtering problem there is implicitly affiliated a LQ control problem.

An understanding of these connections has repeatedly proved useful in cracking interesting applied problems.
For example, Sargent [Sar87] [chs. IX, XIV] and Hansen and Sargent [HS80] formulated and solved control and filtering problems using z-transform methods.

In this lecture we investigate these ideas using mostly elementary linear algebra.

References

Useful references include [Whi63], [HS80], [Orf88], [AP91], and [Mut60].

8.6.2 Infinite Horizon Prediction and Filtering Problems

We pose two related prediction and filtering problems.

We let \( Y_t \) be a univariate \( m^{th} \) order moving average, covariance stationary stochastic process,

\[
Y_t = d(L)u_t
\]  

(8.52)

where \( d(L) = \sum_{j=0}^{m} d_j L^j \), and \( u_t \) is a serially uncorrelated stationary random process satisfying

\[
\mathbb{E}u_t = 0
\]

\[
\mathbb{E}u_t u_s = \begin{cases} 1 & \text{if } t = s \\ 0 & \text{otherwise} \end{cases}
\]  

(8.53)

We impose no conditions on the zeros of \( d(z) \).

A second covariance stationary process is \( X_t \) given by

\[
X_t = Y_t + \varepsilon_t
\]  

(8.54)

where \( \varepsilon_t \) is a serially uncorrelated stationary random process with \( \mathbb{E}\varepsilon_t = 0 \) and \( \mathbb{E}\varepsilon_t \varepsilon_s = 0 \) for all distinct \( t \) and \( s \).

We also assume that \( \mathbb{E}\varepsilon_t \varepsilon_s = 0 \) 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 \( \{X_t, X_{t-1}, \ldots\} \) that minimizes \( \mathbb{E}(\hat{X}_{t+j} - X_{t+j})^2 \).

That is, the problem is to find a \( \gamma_j(L) = \sum_{k=0}^{\infty} \gamma_{jk} L^k \) such that \( \sum_{k=0}^{\infty} |\gamma_{jk}|^2 < \infty \) and \( \mathbb{E}[(\gamma_j(L)X_t - X_{t+j})^2] \) is minimized.

The **linear least squares filtering problem** is to find a \( b(L) = \sum_{j=0}^{\infty} b_j L^j \) such that \( \sum_{j=0}^{\infty} |b_j|^2 < \infty \) and \( \mathbb{E}[b(L)X_t - Y_t]^2 \) is minimized.

Interesting versions of these problems related to the permanent income theory were studied by [Mut60].
**Problem formulation**

These problems are solved as follows

The covariograms of $Y$ and $X$ and their cross covariogram are, respectively,

\[
C_X(\tau) = E X_t X_{t-\tau} \quad C_Y(\tau) = E Y_t Y_{t-\tau} \quad \tau = 0, \pm 1, \pm 2, \ldots \tag{8.55}
\]

The covariance and cross covariance generating functions are defined as

\[
g_X(z) = \sum_{\tau=-\infty}^{\infty} C_X(\tau) z^\tau \\
g_Y(z) = \sum_{\tau=-\infty}^{\infty} C_Y(\tau) z^\tau \tag{8.56}
\]

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, $E v_{1t}^2 = E v_{2t}^2 = 1$, $E v_{1t} v_{2t} = 0$, $E v_{1t} v_{2s} = 0$ for all $t$ and $s$, $E v_{1t} v_{1t-j} = E v_{2t} v_{2t-j} = 0$ for all $j \neq 0$

Let $x_t$ and $y_t$ be two random process given by

\[
y_t = A(L) v_{1t} + B(L) v_{2t} \\
x_t = C(L) v_{1t} + D(L) v_{2t}
\]

Then, as shown for example in [Sar87] [ch. XI], it is true that

\[
g_y(z) = A(z) A(z^{-1}) + B(z) B(z^{-1}) \\
g_x(z) = C(z) C(z^{-1}) + D(z) D(z^{-1}) \tag{8.57}
\]

Applying these formulas to (8.52) – (8.55), we have

\[
g_Y(z) = d(z) d(z^{-1}) \\
g_X(z) = d(z) d(z^{-1}) + h \tag{8.58}
\]

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,

$$X_t = c(L) \eta_t$$  \hfill (8.59)

where $c(L) = \sum_{j=0}^{m} c_j L^j$, with

$$c_0 \eta_t = X_t - \hat{E}[X_t|X_{t-1}, X_{t-2}, \ldots]$$ \hfill (8.60)

Here $\hat{E}$ is the linear least squares projection operator.

Equation (8.60) 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 (8.60) requires that $\eta_t$ lie in the closed linear space spanned by $[X_t, X_{t-1}, \ldots]$. 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 (8.60) that $\eta_t$ is a serially uncorrelated random process, and that a normalization can be imposed so that $\mathbb{E} \eta_t^2 = 1$.

Consequently, an implication of (8.59) is that the covariance generating function of $X_t$ can be expressed as

$$g_X(z) = c(z) c(z^{-1})$$ \hfill (8.61)

It remains to discuss how $c(L)$ is to be computed.

Combining (8.57) and (8.61) gives

$$d(z) d(z^{-1}) + h = c(z) c(z^{-1})$$ \hfill (8.62)

Therefore, we have already showed constructively how to factor the covariance generating function $g_X(z) = d(z) d(z^{-1}) + h$.

We now introduce the **annihilation operator**:

$$\left[ \sum_{j=-\infty}^{\infty} f_j L^j \right]_+ \equiv \sum_{j=0}^{\infty} f_j L^j$$ \hfill (8.63)

In words, $[\cdot ]_+$ means ignore negative powers of $L$.

We have defined the solution of the prediction problem as $\hat{E}[X_{t+j}|X_t, X_{t-1}, \ldots] = \gamma_j(L)X_t$.

Assuming that the roots of $c(z) = 0$ all lie outside the unit circle, the Wiener-Kolmogorov formula for $\gamma_j(L)$ holds:
\[ \gamma_j(L) = \left[ \frac{c(L)}{L^j} \right] + c(L)^{-1} \]  

(8.64)

We have defined the solution of the filtering problem as \( \hat{E}[Y_t \mid X_t, X_{t-1}, \ldots] = b(L)X_t \)

The Wiener-Kolomogorov formula for \( b(L) \) is

\[ b(L) = \left( \frac{g_{yx}(L)}{c(L^{-1})} \right) + c(L)^{-1} \]

or

\[ b(L) = \left[ \frac{d(L)d(L^{-1})}{c(L^{-1})} \right] + c(L)^{-1} \]  

(8.65)

Formulas (8.64) and (8.65) are discussed in detail in [Whi83] and [Sar87]

The interested reader can there find several examples of the use of these formulas in economics Some classic examples using these formulas are due to [Mut60]

As an example of the usefulness of formula (8.65), 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 \( c_0\eta_t = X_t - \hat{E}[X_t \mid X_{t-1}, \ldots], c(L) = \sum_{j=0}^{m} c_jL^j \)

Suppose that at time \( t \), we wish to predict a geometric sum of future \( X \) s, namely

\[ y_t \equiv \sum_{j=0}^{\infty} \delta^jX_{t+j} = \frac{1}{1 - \delta L^{-1}}X_t \]

given knowledge of \( X_t, X_{t-1}, \ldots \)

We shall use (8.65) to obtain the answer

Using the standard formulas (8.57), we have that

\[ g_{yx}(z) = (1 - \delta z^{-1})c(z)c(z^{-1}) \]
\[ g_x(z) = c(z)c(z^{-1}) \]

Then (8.65) becomes

\[ b(L) = \left[ \frac{c(L)}{1 - \delta L^{-1}} \right] + c(L)^{-1} \]  

(8.66)

In order to evaluate the term in the annihilation operator, we use the following result from [HS80]

**Proposition** Let
\( g(z) = \sum_{j=0}^{\infty} g_j z^j \) where \( \sum_{j=0}^{\infty} |g_j|^2 < +\infty \)

\( h(z^{-1}) = (1 - \delta_1 z^{-1}) \cdots (1 - \delta_n z^{-1}), \) where \( |\delta_j| < 1, \) for \( j = 1, \ldots, n \)

Then

\[
\begin{bmatrix} g(z) \\ h(z^{-1}) \end{bmatrix} = \begin{bmatrix} g(z) \\ h(z^{-1}) \end{bmatrix} - \sum_{j=1}^{n} \prod_{k=1 \atop k \neq j}^{n} (\delta_j - \delta_k) \left( \frac{1}{z - \delta_j} \right) \tag{8.67}
\]

and, alternatively,

\[
\begin{bmatrix} g(z) \\ h(z^{-1}) \end{bmatrix} = \sum_{j=1}^{n} B_j \left( \frac{z g(z) - \delta_j g(\delta_j)}{z - \delta_j} \right) \tag{8.68}
\]

where \( B_j = 1/\prod_{k+j}^{n} (1 - \delta_k/\delta_j) \)

Applying formula (8.68) of the proposition to evaluating (8.66) 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

\[
\hat{E} \sum_{j=0}^{\infty} \delta^j X_{t+j} | X_t, x_{t-1}, \ldots = \left[ \frac{1 - \delta c(\delta)L^{-1}c(L)^{-1}}{1 - \delta L^{-1}} \right] X_t \tag{8.69}
\]

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

\[
\mathbb{E}_t \lim_{N \to \infty} \sum_{t=0}^{N} \beta^t \left[ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} [d(L)y_t]^2 \right] \tag{8.70}
\]

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

\[ \eta_t = a_t - E[a_t|a_{t-1}, \ldots] \]

The problem is to maximize (8.70) 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 \( f_a \) is a known sequence. The solution is, from above,

\[ c(L)y_t = c(L)^{-1}a_t \]

or

\[ (1 - \lambda_1 L) \ldots (1 - \lambda_m L)y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} (\lambda_j \beta)^k a_{t+k} \]  

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 (8.69) and (8.71), we have the following solution

\[ (1 - \lambda_1 L) \ldots (1 - \lambda_m L)y_t = \sum_{j=1}^{m} A_j \left[ \frac{1 - \beta \lambda_j c(\beta \lambda_j) L^{-1} c(L)^{-1}}{1 - \beta \lambda_j L^{-1}} \right] a_t \]

8.6.3 Finite Dimensional Prediction

Let \( (x_1, x_2, \ldots, x_T)' = x \) be a \( T \times 1 \) vector of random variables with mean \( \mathbb{E}x = 0 \) and covariance matrix \( \mathbb{E}xx' = V \)

Here \( V \) is a \( T \times T \) positive definite matrix

We shall 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 (8.59)

In this 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 shall be interested in constructing \( j \) step ahead linear least squares predictors of the form

\[ \mathbb{E}[x_T|x_{T-j}, x_{T-j+1}, \ldots, x_1] \]
where $E$ is the linear least squares projection operator

The solution of this problem can be exhibited by first constructing an orthonormal basis of random variables $\varepsilon$ for $x$

Since $V$ is a positive definite and symmetric, we know that there exists a (Cholesky) decomposition of $V$ such that

$$V = L^{-1}(L^{-1})'$$

or

$$LV L' = I$$

where $L$ is lower-trangular, and therefore so is $L^{-1}$

Form the random variable $Lx = \varepsilon$

Then $\varepsilon$ is an orthonormal basis for $x$, since $L$ is nonsingular, and $E\varepsilon\varepsilon' = LEXx'L' = I$

It is convenient to write out the equations $Lx = \varepsilon$ and $L^{-1}\varepsilon = x$

$$(8.72)$$

$$(8.73)$$

We also have

$$(8.74)$$

Notice from $(8.74)$ that $x_t$ is in the space spanned by $\varepsilon_t, \varepsilon_{t-1}, \ldots, \varepsilon_1$, and from $(8.73)$ that $\varepsilon_t$ is in the space spanned by $x_t, x_{t-1}, \ldots, x_1$

Therefore, we have that for $t - 1 \geq m \geq 1$

$$(8.75)$$

For $t - 1 \geq m \geq 1$ rewrite $(8.74)$ as
\[ x_t = \sum_{j=0}^{m-1} L_{t, t-j}^{-1} \varepsilon_{t-j} + \sum_{j=m}^{t-1} L_{t, t-j}^{-1} \varepsilon_{t-j} \]  

(8.76)

Representation (8.76) is an orthogonal decomposition of \( x_t \) into a part \( \sum_{j=m}^{t-1} L_{t, t-j}^{-1} \varepsilon_{t-j} \) that lies in the space spanned by \( [x_{t-m}, x_{t-m+1}, \ldots, x_1] \), and an orthogonal component not in this space.

**Implementation**

Code that computes solutions to LQ control and filtering problems using the methods described here and in *Classical Control with Linear Algebra* can be found in the file `control_and_filter.py`.

Here’s how it looks:

```python
# 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, beta=None):
        
        Parameters
        ----------
        d : list or numpy.array (1-D or a 2-D column vector)
            The order of the coefficients: \([d_0, d_1, \ldots, d_m]\)
        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: \([r_0, r_1, \ldots, r_k]\)
            (optional, if not defined -> deterministic problem)
        beta : 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 up front
#---------------------------------------------
= np.zeros(2 * self.m + 1)
for i in range(- self.m, self.m + 1):
    [self.m - i] = np.sum(np.diag(self.d.reshape(self.m + 1, 1) \ self.d.reshape(1, self.m + 1), k=-i))
    [self.m] = [self.m] + self.h
self. =

# 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
    _r = np.zeros(2 * self.k + 1)
    for i in range(- self.k, self.k + 1):
        _r[self.k - i] = np.sum(np.diag(self.r.reshape(self.k + 1, 1) \ 0 \ self.r.reshape(1, self.k + 1), k=-i))
    if h_eps is None:
        self._r = _r
    else:
        _r[self.k] = _r[0] + h_eps
        self._r = _r

# If \(\beta\) is given, define the transformed variables
#-----------------------------------------------------
if \(\beta\) is None:
    self.\(\beta\) = 1
else:
    self.\(\beta\) = \(\beta\) * (np.arange(self.m + 1)/2) * self.d
    self.y_m = self.y_m * (self.\(\beta\)**(- 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) Constructure 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 $t = 0, 1, \ldots, N-(m+1)$
#----------------------------------------------

W[:] = 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)] =

for i in range(1, m + 1):
    W[N - m + i, -(2 * m + 1 - i):] = [:i]

for i in range(m):
    W_m[N - i, :(m - i)] = [(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 $x_1, x_2, \ldots, x_m$ are the roots then $p(x) = (x - x_1)(x - x_2)\ldots(x - x_m)$

```python
m = self.m
# Calculate the roots of the 2m-polynomial
roots = np.roots()
# sort the roots according to their length (in descending order)
roots_sorted = roots[np.argsort(abs(roots))][::-1]

z_0 = .sum() / np.poly1d(roots, True)(1)
z_1_to_m = roots_sorted[:m]  # we need only those outside the unit circle

# Calculate the roots of the 2m-polynomial
roots = np.roots()
# sort the roots according to their length (in descending order)
roots_sorted = roots[np.argsort(abs(roots))][::-1]

z_0 = .sum() / np.poly1d(roots, True)(1)
z_1_to_m = roots_sorted[:m]  # we need only those outside the unit circle

return z_1_to_m, z_0, lambda
```

def coeffs_of_c(self):
    """
    This function computes the coefficients $\{c_j, j = 0, 1, \ldots, m\}$
    for $c(z) = \sum_{j = 0}^{m} c_j z^j$
    Based on the expression (1.9). The order is
    $\{c_j, j = 0, 1, \ldots, c_{m-1}, c_m\}$
    """

    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 $\{A_{j}, j=1,\ldots,m\}$ and $\{\lambda_{j}, j=1,\ldots,m\}$
    of the expression (1.15)
    """

    lambda_ = 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 - lambda_/lambda[j]
        A[j] = c_0**(-2) / np.prod(denom[np.arange(self.m) != j])

    return lambda_, A

def construct_V(self, N):
```
This function constructs the covariance matrix for $x^N$ (see section 6) for a given period $N$

```python
V = np.zeros((N, N))
_r = self._r
for i in range(N):
    for j in range(N):
        if abs(i-j) <= self.k:
            V[i, j] = _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: $E[\bar{a} | a_t, a_{(t-1)}, ..., a_1, a_0]$

    ```python
    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 $U$

```python
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 $\beta$ is given
#--------------------------------------------
if self.\beta != 1:
    a_hist = a_hist * (self.\beta**(np.arange(N + 1) / 2))[::-1].reshape(N + 1, 1)

a_bar = a_hist - W_m @ self.y_m  # a_bar from lecture

Uy = np.linalg.solve(L, a_bar)  # $U @ y_bar = L^{\bot}$
y_bar = np.linalg.solve(U, Uy)  # $y_bar = U^{\bot}L^{\bot}$

# Reverse the order of y_bar with the matrix $J$
J = np.fliplr(np.eye(N + self.m + 1))
y_hist = J @ np.vstack([y_bar, self.y_m])  # y_hist : concatenated $y_m$ and $y_bar$

#--------------------------------------------
# Transform the optimal sequence back if $\beta$ is given
#--------------------------------------------
if self.\beta != 1:
    y_hist = y_hist * (self.\beta*(- np.arange(-self.m, N + 1)/2)).reshape(N + 1 + self.m, 1)

return y_hist, L, U, y_bar
```

1088 Chapter 8. Time Series Models
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}J
    y_bar = np.linalg.solve(U, Uy) # y_bar = U^{-1}L^{-1}J
    # Reverse the order of y_bar with the matrix J
    J = np.fliplr(np.eye(N + self.m + 1))
    y_hist = J @ np.vstack([y_bar, self.y_m]) # y_hist : concatenated y_m and y_bar
    return y_hist, L, U, y_bar

Let's use this code to tackle two interesting examples

**Example 1**

Consider a stochastic process with moving average representation

\[ x_t = (1 - 2L) \varepsilon_t \]

where \( \varepsilon_t \) is a serially uncorrelated random process with mean zero and variance unity.

We want to use the Wiener-Kolmogorov formula (8.64) to compute the linear least squares forecasts \( \mathbb{E}[x_{t+j} \mid x_t, x_{t-1}, \ldots] \), for \( j = 1, 2 \).

We can do everything we want by setting \( d = r \), generating an instance of \( LQFilter \), then invoking pertinent methods of \( LQFilter \)

```python
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

```python
example.coefs_of_c()
```

```
array([[ 2., -1.]])
```
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 \( V = L^{-1}L^{-1} = LiLi' \) and use it to form the vector of moving average representations \( x = Li \varepsilon \) and the vector of autoregressive representations \( Lx = \varepsilon \)

\[
V = \text{example.construct_V(N=5)}
\]

\[
\begin{bmatrix}
5. & -2. & 0. & 0. & 0. \\
-2. & 5. & -2. & 0. & 0. \\
0. & -2. & 5. & -2. & 0. \\
0. & 0. & -2. & 5. & -2. \\
0. & 0. & 0. & -2. & 5.
\end{bmatrix}
\]

Notice how the lower rows of the moving average representations are converging to the appropriate infinite history Wold representation

\[
\begin{bmatrix}
2.23606798 & 0. & 0. & 0. & 0. \\
-0.89442719 & 2.04939015 & 0. & 0. & 0. \\
0. & -0.97590007 & 2.01186954 & 0. & 0. \\
0. & 0. & -0.99410007 & 2.00293902 & 0. \\
0. & 0. & 0. & -0.99853265 & 2.000733
\end{bmatrix}
\]

Notice how the lower rows of the autoregressive representations are converging to the appropriate infinite history autoregressive representation

\[
L = \text{np.linalg.inv(Li)}
\]

\[
\begin{bmatrix}
0.4472136 & 0. & 0. & 0. & 0. \\
0.19518001 & 0.48795004 & 0. & 0. & 0. \\
0.09467621 & 0.23669053 & 0.49705012 & 0. & 0. \\
0.04698977 & 0.11747443 & 0.2466963 & 0.49926632 & 0. \\
0.02345182 & 0.05862954 & 0.12312203 & 0.24917554 & 0.49981682
\end{bmatrix}
\]

Remark Let \( \pi(z) = \sum_{j=0}^{m} \pi_j z^j \) and let \( z_1, \ldots, z_k \) be the zeros of \( \pi(z) \) that are inside the unit circle, \( k < m \)

Then define

\[
\theta(z) = \pi(z) \left( \frac{z_1 z - 1}{z - z_1} \right) \left( \frac{z_2 z - 1}{z - z_2} \right) \cdots \left( \frac{z_k z - 1}{z - z_k} \right)
\]

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.

**Example 2**

Consider a stochastic process $X_t$ with moving average representation

$$X_t = (1 - \sqrt{2}L^2) \varepsilon_t$$

where $\varepsilon_t$ is a serially uncorrelated random process with mean zero and variance unity.

Let's find a Wold moving average representation for $x_t$.

Let's use the Wiener-Kolomogorov formula (8.64) to compute the linear least squares forecasts $\hat{E}[X_{t+j} | X_{t-1}, \ldots]$ for $j = 1, 2, 3$.

We proceed in the same way as example 1.

```python
m = 2
y_m = np.asarray([[0, 0]]).reshape(m, 1)
d = np.asarray([1, 0, -np.sqrt(2)])
r = np.asarray([1, 0, -np.sqrt(2)])
h = 0.0
example = LQFilter(d, h, y_m, r=d)
example.coeffs_of_c()
array([[ 1.41421356, -0.0, -1.0]])

example.roots_of_characteristic()

(array([[ 1.18920712, -1.18920712]]),
 -1.4142135623731122,
 array([[ 0.84089642, -0.84089642]]))

V = example.construct_V(N=8)
print(V)

[[ 3.00 0.00 -1.41421356 0.00 0.00 0.00 0.00 0.00]
 [ 0.00 3.00 0.00 -1.41421356 0.00 0.00 0.00 0.00]
 [0.00 0.00 3.00 0.00 -1.41421356 0.00 0.00 0.00]
 [-1.41421356 0.00 3.00 0.00 -1.41421356 0.00 0.00 0.00]
 [ 0.00 0.00 0.00 3.00 0.00 -1.41421356 0.00 0.00]
 [0.00 0.00 0.00 -1.41421356 0.00 3.00 0.00 -1.00]
 [0.00 0.00 0.00 0.00 -1.41421356 0.00 3.00 0.00]
 [-1.41421356 0.00 0.00 -1.41421356 0.00 3.00 0.00 -1.00]]
```

8.6. Classical Filtering With Linear Algebra 1091
**Prediction**

It immediately follows from the orthogonality principle of least squares (see [AP91] or [Sar87] [ch. X]) that

$$
\hat{E}[x_t | x_{t-m}, x_{t-m+1}, \ldots x_1] = \sum_{j=m}^{t-1} L_{t-j}^{-1} \varepsilon_{t-j}
$$

$$
= [L_{t,1}^{-1} L_{t,2}^{-1}, \ldots, L_{t,t-m}^{-1} 0 0 \ldots 0] L x
$$

This can be interpreted as a finite-dimensional version of the Wiener-Kolmogorov $m$-step ahead prediction formula.
We can use (8.77) to represent the linear least squares projection of the vector $x$ conditioned on the first $s$ observations $[x_s, x_{s-1}, \ldots, x_1]$

We have

$$
\hat{E}[x \mid x_s, x_{s-1}, \ldots, x_1] = L^{-1} \begin{bmatrix} I_s & 0 \\ 0 & 0_{(t-s)} \end{bmatrix} Lx
$$

(8.78)

This formula will be convenient in representing the solution of control problems under uncertainty.

Equation (8.74) can be recognized as a finite dimensional version of a moving average representation

Equation (8.73) can be viewed as a finite dimension version of an autoregressive representation

Notice that even if the $x_t$ process is covariance stationary, so that $V$ is such that $V_{ij}$ depends only on $|i-j|$, the coefficients in the moving average representation are time-dependent, there being a different moving average for each $t$.

If $x_t$ is a covariance stationary process, the last row of $L^{-1}$ converges to the coefficients in the Wold moving average representation for $\{x_t\}$ as $T \to \infty$.

Further, if $x_t$ is covariance stationary, for fixed $k$ and $j > 0$, $L^{-1}_{T,T-j}$ converges to $L^{-1}_{T-k,T-k-j}$ as $T \to \infty$.

That is, the bottom rows of $L^{-1}$ converge to each other and to the Wold moving average coefficients as $T \to \infty$.

This last observation gives one simple and widely-used practical way of forming a finite $T$ approximation to a Wold moving average representation:

First, form the covariance matrix $E xx' = V$, 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.

8.6.4 Combined Finite Dimensional Control and Prediction

Consider the finite-dimensional control problem, maximize

$$
E \sum_{t=0}^{N} \left\{ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} [d(L)y_t]^2 \right\}, \quad h > 0
$$

where $d(L) = d_0 + d_1 L + \ldots + d_m L^m$, $L$ is the lag operator, $\bar{a} = [a_N, a_{N-1}, \ldots, a_1, a_0]'$ a random vector with mean zero and $E \bar{a} \bar{a}' = V$.

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 feedback-feedforward form

\[ U \mathbf{y} = L^{-1} \tilde{a} + K \begin{bmatrix} y_{-1} \\ \vdots \\ y_{-m} \end{bmatrix} \]

for some \((N + 1) \times m\) matrix \(K\).

Using a version of formula (8.77), we can express \(\hat{E}[\tilde{a} | a_s, a_{s-1}, \ldots, a_0]\) as

\[ \hat{E}[\tilde{a} | a_s, a_{s-1}, \ldots, a_0] = \tilde{U}^{-1} \begin{bmatrix} 0 & 0 \\ 0 & I_{(s+1)} \end{bmatrix} \tilde{U} \tilde{a} \]

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 (8.78) by replacing \(L\) with \(L^T\).

The optimal decision rule to use at time \(0 \leq t \leq N\) is then given by the \((N - t + 1)^{th}\) row of

\[ U \mathbf{y} = L^{-1} \tilde{U}^{-1} \begin{bmatrix} 0 & 0 \\ 0 & I_{(t+1)} \end{bmatrix} \tilde{U} \tilde{a} + K \begin{bmatrix} y_{-1} \\ \vdots \\ y_{-m} \end{bmatrix} \]

### 8.6.5 Exercises

**Exercise 1**

Let \(Y_t = (1 - 2L)u_t\) where \(u_t\) is a mean zero white noise with \(E u_t^2 = 1\). Let

\[ X_t = Y_t + \varepsilon_t \]

where \(\varepsilon_t\) is a serially uncorrelated white noise with \(E \varepsilon_t^2 = 9\), and \(E \varepsilon_t u_s = 0\) for all \(t\) and \(s\).

Find the Wold moving average representation for \(X_t\).

Find a formula for the \(A_{1j}\)s in

\[ \hat{E}X_{t+1} | X_t, X_{t-1}, \ldots = \sum_{j=0}^{\infty} A_{1j} X_{t-j} \]

Find a formula for the \(A_{2j}\)s in

\[ \hat{E}X_{t+2} | X_t, X_{t-1}, \ldots = \sum_{j=0}^{\infty} A_{2j} X_{t-j} \]
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 :math:`\mathbb{E} U_t = 0` for all $t$, $\mathbb{E} U_t U_s' = 0$ for all $s \neq t$, and $\mathbb{E} U_t U_t' = I$ for all $t$

Let $\varepsilon_t$ be an $n \times 1$ vector white noise with mean 0 and contemporaneous covariance matrix $H$, where $H$ is a positive definite matrix

Let $X_t = Y_t + \varepsilon_t$

Define the covariograms as $C_X(\tau) = \mathbb{E} X_t X_{t-\tau}'$, $C_Y(\tau) = \mathbb{E} Y_t Y_{t-\tau}'$, $C_{YX}(\tau) = \mathbb{E} Y_t X_{t-\tau}'$

Then define the matrix covariance generating function, as in (8.47), only interpret all the objects in (8.47) as matrices

Show that the covariance generating functions are given by

$$ g_Y(z) = D(z)D(z^{-1})' $$
$$ g_X(z) = D(z)D(z^{-1})' + H $$
$$ g_{YX}(z) = D(z)D(z^{-1})' $$

A factorization of $g_X(z)$ can be found (see [Roz67] or [Whi83]) of the form

$$ D(z)D(z^{-1})' + H = C(z)C(z^{-1})', \quad C(z) = \sum_{j=0}^{m} C_j z^j $$

where the zeros of $|C(z)|$ do not lie inside the unit circle

A vector Wold moving average representation of $X_t$ is then

$$ X_t = C(L)\eta_t $$

where $\eta_t$ is an $(n \times 1)$ vector white noise that is fundamental for $X_t$

That is, $X_t - \mathbb{E} [X_t | X_{t-1}, X_{t-2} \ldots] = C_0 \eta_t$

The optimum predictor of $X_{t+j}$ is

$$ \mathbb{E} [X_{t+j} | X_t, X_{t-1}, \ldots] = \left[ \frac{C(L)}{L^J} \right]_+ \eta_t $$

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

$$ \mathbb{E} [X_{t+j} | X_t, X_{t-1}, \ldots] = \left[ \frac{C(L)}{L^J} \right]_+ C(L)^{-1} X_t $$
Here we look at models in which a value function for one Bellman equation has as an argument the value function for another Bellman equation.

9.1 Dynamic Stackelberg Problems

Contents

- Dynamic Stackelberg Problems
  - Overview
  - The Stackelberg Problem
  - Solving the Stackelberg Problem
  - Shadow prices
  - A Large Firm With a Competitive Fringe
  - Another Example
  - Concluding Remarks
  - Exercises

9.1.1 Overview

Previous lectures including *LQ dynamic programming*, *rational expectations equilibrium*, and *Markov perfect equilibrium* lectures have studied decision problems that are recursive in what we can call natural state variables, such as

- stocks of capital (fiscal, financial and human)
- wealth
- information that helps forecast future prices and quantities that impinge on future payoffs
Optimal decision rules are functions of the natural state variables in problems that are recursive in the natural state variables.

In this lecture, we describe problems that are not recursive in the natural state variables.

Kydland and Prescott [KP77], [Pre77] and Calvo [Cal78] gave examples of such decision problems.

These problems have the following features:

- Time $t \geq 0$ actions of decision makers called followers depend on time $s \geq t$ decisions of another decision maker called a Stackelberg leader.
- At time $t = 0$, the Stackelberg leader chooses his actions for all times $s \geq 0$.
- In choosing actions for all times at time 0, the Stackelberg leader can be said to commit to a plan.
- The Stackelberg leader has distinct optimal decision rules at time $t = 0$, on the one hand, and at times $t \geq 1$, on the other hand.
- The Stackelberg leader's decision rules for $t = 0$ and $t \geq 1$ have distinct state variables.
- Variables that encode history dependence appear in optimal decision rules of the Stackelberg leader at times $t \geq 1$.
- These properties of the Stackelberg leader's decision rules are symptoms of the time inconsistency of optimal government plans.

An example of a time inconsistent optimal rule is that of a

- a large agent (e.g., a government) that confronts a competitive market composed of many small private agents, and in which
  - private agents' decisions at each date are influenced by their forecasts of the large agent's future actions.

The rational expectations equilibrium concept plays an essential role.

A rational expectations restriction implies that when it chooses its future actions, the Stackelberg leader also chooses the followers' expectations about those actions.

The Stackelberg leader understands and exploits that situation.

In a rational expectations equilibrium, the Stackelberg leader's time $t$ actions confirm private agents forecasts of those actions.

The requirement to confirm prior followers' forecasts puts constraints on the Stackelberg leader's time $t$ decisions that prevent its problem from being recursive in natural state variables.

These additional constraints make the Stackelberg leader's decision rule at $t$ depend on the entire history of the natural state variables from time 0 to time $t$.

This lecture displays these principles within the tractable framework of linear quadratic problems.

It is based on chapter 19 of [LS18].
9.1.2 The Stackelberg Problem

We use the optimal linear regulator (a.k.a. the linear-quadratic dynamic programming problem described in *LQ Dynamic Programming problems*) to solve a linear quadratic version of what is known as a dynamic Stackelberg problem.

For now we refer to the Stackelberg leader as the government and the Stackelberg follower as the representative agent or private sector.

Soon we will give an application with another interpretation of these two decision makers.

Let $z_t$ be an $n_x \times 1$ 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 $t$.

Let $u_t$ be a vector of government instruments.

The $z_t$ vector is inherited physically from the past.

But $x_t$ is inherited as a consequence of decisions made by the Stackelberg planner at time $t = 0$.

Included in $x_t$ might be prices and quantities that adjust instantaneously to clear markets at time $t$.

Let $y_t = \begin{bmatrix} z_t \\ x_t \end{bmatrix}$

Define the government's one-period loss function

$$r(y, u) = y' R y + u' Q u$$

Subject to an initial condition for $z_0$, but not for $x_0$, a government wants to maximize

$$-\sum_{t=0}^{\infty} \beta^t r(y_t, u_t)$$

The government makes policy in light of the model

$$\begin{bmatrix} I & 0 \\ G_{21} & G_{22} \end{bmatrix} \begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \hat{A}_{11} & \hat{A}_{12} \\ \hat{A}_{21} & \hat{A}_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + \hat{B} u_t$$

We assume that the matrix on the left is invertible, so that we can multiply both sides of (9.3) by its inverse to obtain

$$\begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + B u_t$$

or

1 The problem assumes that there are no cross products between states and controls in the return function. A simple transformation converts a problem whose return function has cross products into an equivalent problem that has no cross products. For example, see [HS08] (chapter 4, pp. 72-73).
\[ y_{t+1} = Ay_t + Bu_t \]  

(9.5)

The private sector's behavior is summarized by the second block of equations of (9.4) or (9.5). These equations typically include the first-order conditions of private agents' optimization problem (i.e., their Euler equations).

These Euler equations summarize the forward-looking aspect of private agents' behavior and express how their time \( t \) decisions depend on government actions at times \( s \geq t \).

When combined with a stability condition to be imposed below, the Euler equations summarize the private sectors' best response to the sequence of actions by the government.

The government maximizes (9.2) by choosing sequences \( \{u_t, x_t, z_{t+1}\}_{t=0}^{\infty} \) subject to (9.5) 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 government uses its understanding of the responses restricted by (9.5) to manipulate private sector actions.

To indicate the features of the Stackelberg leaders problem that make \( x_t \) a vector of forward-looking variables, write the second block of system (9.3) as

\[ x_t = \phi_1 z_t + \phi_2 z_{t+1} + \phi_3 u_t + \phi_0 x_{t+1}, \]  

(9.6)

where \( \phi_0 = \hat{A}_{22}^{-1} G_{22} \).

The models we study in this chapter typically satisfy

**Forward-Looking Stability Condition** The eigenvalues of \( \phi_0 \) are bounded in modulus by \( \beta^{-5} \).

This stability condition makes equation (9.6) explosive if solved backwards but stable if solved forwards.

See the appendix of chapter 2 of [LS18]

So we solve equation (9.6) forward to get

\[ x_t = \sum_{j=0}^{\infty} \phi_0^j [\phi_1 z_{t+j} + \phi_2 z_{t+j+1} + \phi_3 u_{t+j}] . \]  

(9.7)

In choosing \( u_t \) for \( t \geq 1 \) at time 0, the government takes into account how future \( z \) and \( u \) affect earlier \( x \) through equation (9.7).

The lecture on history dependent policies analyzes an example about *Ramsey taxation* in which, as is typical in such problems, the last \( n_x \) equations of (9.4) or (9.5) constitute *implementability constraints* that are formed by the Euler equations of a competitive fringe or private sector.

A *certainty equivalence principle* allows us to work with a nonstochastic model (see *LQ dynamic programming*).
That is, we would attain the same decision rule if we were to replace \( x_{t+1} \) with the forecast \( E_t x_{t+1} \) and to add a shock process \( C \epsilon_{t+1} \) to the right side of (9.5), where \( \epsilon_{t+1} \) is an IID random vector with mean zero and identity covariance matrix

Let \( s^t \) denote the history of any variable \( s \) from 0 to \( t \)

[MS85], [HR85], [PL92], [Sar87], [Pea92], and others have all studied versions of the following problem:

**Problem S:** The Stackelberg problem is to maximize (9.2) by choosing an \( x_0 \) and a sequence of decision rules, the time \( t \) component of which maps a time \( t \) history of the natural state \( z^t \) into a time \( t \) decision \( u_t \) of the Stackelberg leader

The Stackelberg leader chooses this sequence of decision rules once and for all at time \( t = 0 \)

Another way to say this is that he commits to this sequence of decision rules at time 0

The maximization is subject to a given initial condition for \( z_0 \)

But \( x_0 \) is among the objects to be chosen by the Stackelberg leader

The optimal decision rule is history dependent, meaning that \( u_t \) depends not only on \( z_t \) but at \( t \geq 1 \) also on lags of \( z \)

History dependence has two sources: (a) the government’s ability to commit\(^2\) to a sequence of rules at time 0 as in the lecture on history dependent policies, and (b) the forward-looking behavior of the private sector embedded in the second block of equations (9.4) as exhibited by (9.7)

### 9.1.3 Solving the Stackelberg Problem

#### Some Basic Notation

For any vector \( a_t \), define \( \vec{a}_t = [a_t, a_{t+1}, \ldots] \).

Define a feasible set of \((\vec{y}_1, \vec{u}_0)\) sequences

\[
\Omega(y_0) = \{ (\vec{y}_1, \vec{u}_0) : y_{t+1} = Ay_t + Bu_t, \forall t \geq 0 \}
\]

Note that in the definition of \( \Omega(y_0) \), \( y_0 \) is taken as given.

Eventually, the \( x_0 \) component of \( y_0 \) will be chosen, though it is taken as given in \( \Omega(y_0) \)

#### 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 \( t \geq 1 \)

Subproblem 2 is solved the **Stackelberg leader** at \( t = 0 \)

\(^2\) The government would make different choices were it to choose sequentially, that is, were it to select its time \( t \) action at time \( t \). See the lecture on history dependent policies

9.1. Dynamic Stackelberg Problems 1101
Subproblem 1

\[ v(y_0) = \max_{(\bar{y}, \bar{u}) \in \Omega(y_0)} - \sum_{t=0}^{\infty} \beta^t r(y_t, u_t) \]  
(9.8)

Subproblem 2

\[ w(z_0) = \max_{x_0} v(y_0) \]  
(9.9)

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, \( z_0 \).

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

\[ v(y) = \max_{u, y^*} \{-r(y, u) + \beta v(y^*)\} \]  
(9.10)

where the maximization is subject to

\[ y^* = Ay + Bu \]  
(9.11)

and \( y^* \) denotes next periods value.

Substituting \( v(y) = -y'Py \) into Bellman equation (9.10) gives

\[ -y'Py = \max_{u, y^*} \{-y'Ry - u'Qu - \beta y'^*Py^*\} \]

which as in lecture linear regulator gives rise to the algebraic matrix Riccati equation

\[ P = R + \beta A'PA - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA \]  
(9.12)
and the optimal decision rule coefficient vector

$$F = \beta (Q + \beta B'PB)^{-1} B'PA,$$

(9.13)

where the optimal decision rule is

$$u_t = -Fy_t.$$  

(9.14)

Subproblem 2

The value function $v(y_0)$ satisfies

$$v(y_0) = -z_0'P_{11}z_0 - 2x_0'P_{21}z_0 - x_0'P_{22}x_0$$

(9.15)

where

$$P = \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix}$$

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

$$x_0 = -P_{22}^{-1}P_{21}z_0.$$  

(9.16)

Summary

We solve the Stackelberg problem by

- formulating a particular optimal linear regulator
- solving the associated matrix Riccati equation (9.12) for $P$
- computing $F$
- then partitioning $P$ to obtain representation (9.16)
**Manifestation of time inconsistency**

We have seen that for \( t \geq 0 \) the optimal decision rule for the Stackelberg leader has the form

\[
u_t = -F y_t
\]

or

\[
u_t = f_{11} z_t + f_{12} x_t
\]

where for \( t \geq 1 \), \( x_t \) is effectively a state variable, albeit not a *natural* one, inherited from the past.

The means that for \( t \geq 1 \), \( x_t \) is *not* a function of \( z_t \) only (though it is at \( t = 0 \)) and that \( x_t \) exerts an independent influence on \( u_t \).

The situation is different at \( t = 0 \).

For \( t = 0 \), the optimal choice of \( x_0 = -P_{22}^{-1} P_{21} z_0 \) described in equation (9.16) implies that

\[
u_0 = (f_{11} - f_{12} P_{22}^{-1} P_{21}) z_0
\]

(9.17)

So for \( t = 0 \), \( u_0 \) is a linear function of the natural state variable \( z_0 \) only.

But for \( t \geq 0 \), \( x_t \neq -P_{22}^{-1} P_{21} z_t \).

Nor does \( x_t \) equal any other linear combination of \( z_t \) for \( t \geq 1 \).

This means that \( x_t \) has an independent role in shaping \( u_t \) for \( t \geq 1 \).

All of this means that the Stackelberg leaders decision rule at \( t \geq 1 \) differs from its decision rule at \( t = 0 \).

As indicated at the beginning of this lecture, this difference is a symptom of the *time inconsistency* of the optimal Stackelberg plan.

**9.1.4 Shadow prices**

The history dependence of the governments plan can be expressed in the dynamics of Lagrange multipliers \( \mu_x \) on the last \( n_x \) equations of (9.3) or (9.4).

These multipliers measure the cost today of honoring past government promises about current and future settings of \( u \).

We shall soon show that as a result of optimally choosing \( x_0 \), it is appropriate to initialize the multipliers to zero at time \( t = 0 \).

This is true because at \( t = 0 \), there are no past promises about \( u \) to honor.

But the multipliers \( \mu_x \) take nonzero values thereafter, reflecting future costs to the government of confirming the private sectors earlier expectations about its time \( t \) actions.

From the *linear regulator* lecture, the formula \( \mu_t = P y_t \) for the vector of shadow prices on the transition equations is

\[
\mu_t = \begin{bmatrix}
\mu_{zt} \\
\mu_{xt}
\end{bmatrix}
\]
The shadow price $\mu_{xt}$ on the forward-looking variables $x_t$ evidently equals

$$
\mu_{xt} = P_{21}z_t + P_{22}x_t. 
$$

(9.18)

So (9.16) is equivalent with

$$
\mu_{x0} = 0.
$$

(9.19)

### 9.1.5 A Large Firm With a Competitive Fringe

As an example, this section studies the equilibrium of an industry with a large firm that acts as a Stackelberg leader with respect to a competitive fringe.

Sometimes the large firm is called the monopolist even though there are actually many firms in the industry.

The industry produces a single nonstorable homogeneous good, the quantity of which is chosen in the previous period.

One large firm produces $Q_t$ and a representative firm in a competitive fringe produces $q_t$.

The representative firm in the competitive fringe acts as a price taker and chooses sequentially.

The large firm commits to a policy at time 0, taking into account its ability to manipulate the price sequence, both directly through the effects of its quantity choices on prices, and indirectly through the responses of the competitive fringe to its forecasts of prices.

The costs of production are $C_t = e Q_t + .5g Q_t^2 + .5c (Q_{t+1} - Q_t)^2$ for the large firm and $\sigma_t = dq_t + .5h q_t^2 + .5c (q_{t+1} - q_t)^2$ for the competitive firm, where $d > 0, e > 0, c > 0, g > 0, h > 0$ are cost parameters.

There is a linear inverse demand curve

$$
p_t = A_0 - A_1 (Q_t + \bar{q}_t) + v_t,
$$

(9.20)

where $A_0, A_1$ are both positive and $v_t$ is a disturbance to demand governed by

$$
v_{t+1} = \rho v_t + C_s \epsilon_{t+1}
$$

(9.21)

where $|\rho| < 1$ and $\epsilon_{t+1}$ is an IID sequence of random variables with mean zero and variance 1.

In (9.20), $\bar{q}_t$ is equilibrium output of the representative competitive firm.

In equilibrium, $\bar{q}_t = q_t$, but we must distinguish between $q_t$ and $\bar{q}_t$ in posing the optimum problem of a competitive firm.

5 [HS08] (chapter 16), uses this model as a laboratory to illustrate an equilibrium concept featuring robustness in which at least one of the agents has doubts about the stochastic specification of the demand shock process.

---

**9.1. Dynamic Stackelberg Problems**

1105
The competitive fringe

The representative competitive firm regards \( \{p_t\}_{t=0}^{\infty} \) as an exogenous stochastic process and chooses an output plan to maximize

\[
E_0 \sum_{t=0}^{\infty} \beta^t \{p_t q_t - \sigma_t\}, \quad \beta \in (0, 1)
\]  

subject to \( q_0 \) given, where \( E_t \) is the mathematical expectation based on time \( t \) information.

Let \( i_t = q_{t+1} - q_t \)

We regard \( i_t \) as the representative firms control at \( t \)

The first-order conditions for maximizing (9.22) are

\[
i_t = E_t \beta i_{t+1} - c^{-1} \beta h q_{t+1} + c^{-1} \beta E_t (p_{t+1} - d)
\]  

for \( t \geq 0 \)

We appeal to a certainty equivalence principle to justify working with a non-stochastic version of (9.23) formed by dropping the expectation operator and the random term \( \epsilon_{t+1} \) from (9.21)

We use a method of \([\text{Sar79}]\) and \([\text{Tow83}]\) \(^4\)

We shift (9.20) forward one period, replace conditional expectations with realized values, use (9.20) to substitute for \( p_{t+1} \) in (9.23), and set \( q_t = \overline{q}_t \) and \( i_t = \overline{i}_t \) for all \( t \geq 0 \) to get

\[
\overline{i}_t = \beta \overline{i}_{t+1} - c^{-1} \beta h \overline{q}_{t+1} + c^{-1} \beta (A_0 - d) - c^{-1} \beta A_1 \overline{q}_{t+1} - c^{-1} \beta A_1 Q_{t+1} + c^{-1} \beta v_{t+1}
\]  

(9.24)

Given sufficiently stable sequences \( \{Q_t, v_t\} \), we could solve (9.24) and \( \overline{i}_t = \overline{q}_{t+1} - \overline{q}_t \) to express the competitive fringes output sequence as a function of the (tail of the) monopolists output sequence

(This would be a version of representation (9.7))

It is this feature that makes the monopolists problem fail to be recursive in the natural state variables \( \overline{q}, Q \)

The monopolist arrives at period \( t > 0 \) facing the constraint that it must confirm the expectations about its time \( t \) decision upon which the competitive fringe based its decisions at dates before \( t \)

The monopolists problem

The monopolist views the sequence of the competitive firms Euler equations as constraints on its own opportunities

They are implementability constraints on the monopolists choices

\(^4\) They used this method to compute a rational expectations competitive equilibrium. Their key step was to eliminate price and output by substituting from the inverse demand curve and the production function into the firms first-order conditions to get a difference equation in capital.
Including the implementability constraints, we can represent the constraints in terms of the transition law facing the monopolist:

$$
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
A_0 - d & 1 - A_1 & - A_1 - h & c
\end{bmatrix}
\begin{bmatrix}
v_{t+1} \\
Q_{t+1} \\
\overline{q}_{t+1} \\
\overline{q}_t \\
\overline{q}_t
\end{bmatrix}
= 
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & \rho & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & \frac{c}{\beta} \\
0 & 0 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
v_t \\
Q_t \\
\overline{q}_t \\
\overline{q}_t \\
\overline{q}_t
\end{bmatrix}
+ 
\begin{bmatrix}
0 \\
1 \\
0 \\
0 \\
0
\end{bmatrix}

u_t
$$

(9.25)

where $u_t = Q_{t+1} - Q_t$ is the control of the monopolist at time $t$.

The last row portrays the implementability constraints (9.24).

Represent (9.25) as

$$
y_{t+1} = Ay_t + Bu_t
$$

(9.26)

Although we have included the competitive fringes choice variable $\overline{q}_t$ as a component of the state $y_t$ in the monopolists transition law (9.26), $\overline{q}_t$ is actually a jump variable.

Nevertheless, the analysis above implies that the solution of the large firms problem is encoded in the Riccati equation associated with (9.26) as the transition law.

Let's decode it.

To match our general setup, we partition $y_t$ as $y_t' = [z_t' \ x_t']$ where $z_t' = [1 \ v_t \ Q_t \ \overline{q}_t]$ and $x_t = \overline{q}_t$.

The monopolists problem is

$$
\max_{\{u_t, p_t, Q_t, \overline{q}_t, z_t\}} \sum_{t=0}^{\infty} \beta^t \{ p_t Q_t - C_t \}
$$

subject to the given initial condition for $z_0$, equations (9.20) and (9.24) and $\overline{q}_t = \overline{q}_{t+1} - \overline{q}_t$, as well as the laws of motion of the natural state variables $z$.

Notice that the monopolist in effect chooses the price sequence, as well as the quantity sequence of the competitive fringe, albeit subject to the restrictions imposed by the behavior of consumers, as summarized by the demand curve (9.20) and the implementability constraint (9.24) that describes the best responses of firms in the competitive fringe.

By substituting (9.20) into the above objective function, the monopolists problem can be expressed as

$$
\max_{\{u_t\}} \sum_{t=0}^{\infty} \beta^t \{ (A_0 - A_1 (\overline{q}_t + Q_t) + v_t) Q_t - eQ_t - .5gQ_t^2 - .5cu_t^2 \}
$$

(9.27)

subject to (9.26)

This can be written.

### 9.1. Dynamic Stackelberg Problems
\[
\max_{\{u_t\}} - \sum_{t=0}^{\infty} \beta^t \left\{ y_t' R y_t + u_t' Q u_t \right\} \tag{9.28}
\]

subject to (9.26) where

\[
R = \begin{bmatrix}
0 & 0 & \frac{A_0 - \epsilon}{2} & 0 & 0 \\
0 & 0 & -\frac{A_1}{2} & -\frac{A_2}{2} & 0 \\
0 & 0 & \frac{A_0 - \epsilon}{2} & 0 & 0 \\
0 & 0 & -\frac{A_1}{2} & 0 & 0
\end{bmatrix}
\]

and \( Q = \frac{\epsilon}{2} \)

Under the Stackelberg plan, \( u_t = -F y_t \), which implies that the evolution of \( y \) under the Stackelberg plan as

\[
\overline{y}_{t+1} = (A - BF) \overline{y}_t \tag{9.29}
\]

where \( \overline{y}_t = \begin{bmatrix} 1 & v_t & Q_t & \overline{q}_t & \overline{\tau}_t \end{bmatrix}' \)

**Recursive formulation of a followers problem**

We now make use of a Big \( K \), little \( k \) trick (see rational expectations equilibrium) to formulate a recursive version of a followers problem cast in terms of an ordinary Bellman equation.

The individual firm faces \( \{p_t\} \) as a price taker and believes

\[
p_t = a_0 - a_1 Q_t - a_1 \overline{q}_t + v_t
\equiv E_p \left[ \overline{y}_t \right] \tag{9.30}
\]

(Please remember that \( \overline{q}_t \) is a component of \( \overline{y}_t \))

From the point of the view of a representative firm in the competitive fringe, \( \{\overline{y}_t\} \) is an exogenous process.

A representative fringe firm wants to forecast \( \overline{y} \) because it wants to forecast what it regards as the exogenous price process \( \{p_t\} \).

Therefore it wants to forecast the determinants of future prices

- future values of \( Q \) and
- future values of \( \overline{q} \)

An individual follower firm confronts state \( \begin{bmatrix} \overline{y}_t & q_t \end{bmatrix}' \) where \( q_t \) is its current output as opposed to \( \overline{q} \) within \( \overline{y} \).

It believes that it chooses future values of \( q_t \) but not future values of \( \overline{q}_t \).

(This is an application of a Big \( K \), little \( k \) idea)

The follower faces law of motion
\[
\begin{bmatrix}
\bar{y}_{t+1} \\
q_{t+1}
\end{bmatrix} = \begin{bmatrix}
A - BF & 0 \\
0 & 1
\end{bmatrix} \begin{bmatrix}
\bar{y}_t \\
q_t
\end{bmatrix} + \begin{bmatrix}
0 \\
1
\end{bmatrix} i_t
\]

(9.31)

We calculated \( F \) and therefore \( A - BF \) earlier.

We can restate the optimization problem of the representative competitive firm.

The firm takes \( \bar{y}_t \) as an exogenous process and chooses an output plan \( \{q_t\} \) to maximize

\[
E_0 \sum_{t=0}^{\infty} \beta^t \{ p_t q_t - \sigma_t \}, \quad \beta \in (0, 1)
\]

subject to \( q_0 \) given the law of motion (9.29) and the price function (9.30) and where the costs are still \( \sigma_t = dq_t + .5hq_t^2 + .5c(q_{t+1} - q_t)^2 \)

The representative firms problem is a linear-quadratic dynamic programming problem with matrices \( A_s, B_s, Q_s, R_s \) that can be constructed easily from the above information.

The representative firms decision rule can be represented as

\[
i_t = -F_s \begin{bmatrix}
1 \\
v_t \\
Q_t \\
\bar{q}_t \\
i_t \\
q_t
\end{bmatrix}
\]

(9.32)

Now let's stare at the decision rule (9.32) for \( i_t \), apply Big \( K \), little \( k \) logic again, and ask what we want in order to verify a recursive representation of a representative followers choice problem.

- We want decision rule (9.32) to have the property that \( i_t = \bar{i}_t \) when we evaluate it at \( q_t = \bar{q}_t \)

We inherit these desires from a Big \( K \), little \( k \) logic.

Here we apply a Big \( K \), little \( k \) logic in two parts to make the representative firm be representative after solving the representative firms optimization problem.

- We want \( q_t = \bar{q}_t \)
- We want \( i_t = \bar{i}_t \)

**Numerical example**

We computed the optimal Stackelberg plan for parameter settings \( A_0, A_1, \rho, C, c, d, d, e, g, h, \beta = 100, 1, .8, .2, 1, 20, 20, .2, .2, .95 \)

\(^5\) These calculations were performed by functions located in `dyn_stack/oligopoly.py`. 

---

**9.1. Dynamic Stackelberg Problems**

1109
For these parameter values the monopolists decision rule is

\[
\begin{bmatrix}
1 \\
v_t \\
Q_t \\
\bar{q}_t \\
t_t
\end{bmatrix} = \begin{bmatrix}
83.98 & 0.78 & -0.95 & -1.31 & -2.07 \\
\end{bmatrix}
\]

for \( t \geq 0 \)

and

\[
x_0 \equiv \bar{\bar{u}}_0 = \begin{bmatrix}
31.08 & 0.29 & -0.15 & -0.56 \\
v_0 \\
Q_0 \\
\bar{q}_0 \\
t_0
\end{bmatrix}
\]

For this example, starting from \( z_0 = \begin{bmatrix} 1 & v_0 & Q_0 & \bar{q}_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 25 & 46 \end{bmatrix} \), the monopolist chooses to set \( i_0 = 1.43 \)

That choice implies that

- \( i_1 = 0.25 \), and
- \( z_1 = \begin{bmatrix} 1 & v_1 & Q_1 & \bar{q}_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 21.83 & 47.43 \end{bmatrix} \)

A monopolist who started from the initial conditions \( \tilde{z}_0 = z_1 \) would set \( i_0 = 1.10 \) instead of .25 as called for under the original optimal plan.

The preceding little calculation reflects the time inconsistency of the monopolists optimal plan.

The recursive representation of the decision rule for a representative fringe firm is

\[
i_t = \begin{bmatrix}
0 & 0 & 0.34 & 1 & -0.34 \\
v_t \\
Q_t \\
\bar{q}_t \\
t_t
\end{bmatrix}
\]

which we have computed by solving the appropriate linear-quadratic dynamic programming problem described above.

Notice that, as expected, \( i_t = \bar{\bar{i}}_t \) when we evaluate this decision rule at \( q_t = \bar{q}_t \).

### 9.1.6 Another Example

Please see *Ramsey plans, time Inconsistency, sustainable Plans* for a Stackelberg plan computed using methods described here.
9.1.7 Concluding Remarks

This lecture is our first encounter with a class of problems in which optimal decision rules are history dependent.\(^6\)

We shall encounter other examples in lectures optimal taxation with state-contingent debt and optimal taxation without state-contingent debt.

Many more examples of such problems are described in chapters 20-24 of [LS18].

9.1.8 Exercises

Exercise 1

There is no uncertainty.

For \( t \geq 0 \), a monetary authority sets the growth of (the log of) money according to

\[
m_{t+1} = m_t + u_t \tag{9.33}
\]

subject to the initial condition \( m_0 > 0 \) given.

The demand for money is

\[
m_t - p_t = -\alpha(p_{t+1} - p_t) \tag{9.34}
\]

where \( \alpha > 0 \) and \( p_t \) is the log of the price level.

Equation (9.33) can be interpreted as the Euler equation of the holders of money.

a. Briefly interpret how (9.33) makes the demand for real balances vary inversely with the expected rate of inflation. Temporarily (only for this part of the exercise) drop (9.33) and assume instead that \( \{m_t\} \) is a given sequence satisfying \( \sum_{t=0}^{\infty} m_t^2 < +\infty \). Solve the difference equation (9.33) forward to express \( p_t \) as a function of current and future values of \( m_s \). Note how future values of \( m \) influence the current price level.

At time 0, a monetary authority chooses (commits to) a possibly history-dependent strategy for setting \( \{u_t\}_{t=0}^{\infty} \).

The monetary authority orders sequences \( \{m_t, p_t\}_{t=0}^{\infty} \) according to

\[
- \sum_{t=0}^{\infty} .95^t \left[ (p_t - \bar{p})^2 + u_t^2 + .00001m_t^2 \right] \tag{9.35}
\]

Assume that \( m_0 = 10 \), \( \alpha = 5 \), \( \bar{p} = 1 \).

---

\(^6\) For another application of the techniques in this lecture and how they related to the method recommended by [KP80b], please see this lecture.
b. Please briefly interpret this problem as one where the monetary authority wants to stabilize the price level, subject to costs of adjusting the money supply and some implementability constraints. (We include the term \(0.00001 m_t^2\) for purely technical reasons that you need not discuss.)

c. Please write and run a Python program to find the optimal sequence \(\{u_t\}_{t=0}^\infty\)

d. Display the optimal decision rule for \(u_t\) as a function of \(u_{t-1}, m_t, m_{t-1}\)

e. Compute the optimal \(\{m_t, p_t\}_t\) sequence for \(t = 0, \ldots, 10\)

Hints:
- The optimal \(\{m_t\}\) sequence must satisfy \(\sum_{t=0}^\infty (.95)^t m_t^2 < +\infty\)
- Code can be found in the file `lqcontrol.py` from the QuantEcon.py package that implements the optimal linear regulator

Exercise 2

A representative consumer has quadratic utility functional

\[
\sum_{t=0}^\infty \beta^t \{-0.5(b - c_t)^2\}
\]

(9.36)

where \(\beta \in (0, 1)\), \(b = 30\), and \(c_t\) is time \(t\) consumption

The consumer faces a sequence of budget constraints

\[
c_t + a_{t+1} = (1 + r)a_t + y_t - \tau_t
\]

(9.37)

where
- \(a_t\) is the household's holdings of an asset at the beginning of \(t\)
- \(r > 0\) is a constant net interest rate satisfying \(\beta(1 + r) < 1\)
- \(y_t\) is the consumer's endowment at \(t\)

The consumer's plan for \((c_t, a_{t+1})\) has to obey the boundary condition \(\sum_{t=0}^\infty \beta^t a_t^2 < +\infty\)

Assume that \(y_0, a_0\) are given initial conditions and that \(y_t\) obeys

\[
y_t = \rho y_{t-1}, \quad t \geq 1,
\]

(9.38)

where \(|\rho| < 1\). Assume that \(a_0 = 0\), \(y_0 = 3\), and \(\rho = .9\)

At time 0, a planner commits to a plan for taxes \(\{\tau_t\}_{t=0}^\infty\)

The planner designs the plan to maximize
\[
\sum_{t=0}^{\infty} \beta^t \left\{ -0.5(c_t - b)^2 - \tau_t^2 \right\} (9.39)
\]

over \(\{c_t, \tau_t\}_{t=0}^{\infty}\) subject to the implementability constraints in (9.37) for \(t \geq 0\) and

\[
\lambda_t = \beta(1 + r)\lambda_{t+1} (9.40)
\]

for \(t \geq 0\), where \(\lambda_t \equiv (b - c_t)\)

a. Argue that (9.40) is the Euler equation for a consumer who maximizes (9.36) subject to (9.37), taking \(\{\tau_t\}\) as a given sequence

b. Formulate the planners problem as a Stackelberg problem

c. For \(\beta = 0.95, b = 30, \beta(1 + r) = 0.95\), formulate an artificial optimal linear regulator problem and use it to solve the Stackelberg problem

d. Give a recursive representation of the Stackelberg plan for \(\tau_t\)

### 9.2 Ramsey plans, Time Inconsistency, Sustainable Plans

**Contents**

- Ramsey plans, Time Inconsistency, Sustainable Plans
  - Overview
  - The Model
  - Structure
  - Intertemporal Influences
  - Four Models of Government Policy
  - A Ramsey Planner
  - A Constrained-to-a-Constant-\(\mu\) Ramsey Government
  - Markov Perfect Governments
  - Equilibrium Outcomes for Three Models of Government Policy Making
  - A Fourth Model of Government Decision Making
  - Sustainable or Credible plan
  - Comparison of Equilibrium Values
  - Note on Dynamic Programming Squared

**Co-author: Sebastian Graves**
9.2.1 Overview

This lecture describes a linear-quadratic version of a model that Guillermo Calvo [Cal78] used to illustrate the time inconsistency of optimal government plans. Like Chang [Cha98], we use the model as a laboratory in which to explore 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 planner.

A theme of this lecture is that timing protocols affect outcomes. We will use ideas from papers by Cagan [Cag56], Calvo [Cal78], Stokey [Sto89], [Sto91], Chari and Kehoe [CK90], Chang [Cha98], and Abreu [Abr88] as well as from chapter 19 of [LS18].

In addition, we will 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.

9.2.2 The Model

There is no uncertainty.

Let:

- \( p_t \) be the log of the price level
- \( m_t \) be the log of nominal money balances
- \( \theta_t = p_{t+1} - p_t \) be the rate of inflation between \( t \) and \( t + 1 \)
- \( \mu_t = m_{t+1} - m_t \) be the rate of growth of nominal balances.

The demand for real balances is governed by a perfect foresight version of the Cagan [Cag56] demand function:
\[ m_t - p_t = -\alpha (p_{t+1} - p_t) , \quad \alpha > 0 \]  
(9.41)

for \( t \geq 0 \)

Equation (9.41) asserts that the demand for real balances is inversely related to the publics expected rate of inflation, which here equals the actual rate of inflation 

(When there is no uncertainty, an assumption of rational expectations simplifies to perfect foresight) 

(See [Sar77] for a rational expectations version of the model when there is uncertainty)

Subtracting the demand function at time \( t \) from the demand function at \( t + 1 \) gives:

\[ \mu_t - \theta_t = -\alpha \theta_{t+1} + \alpha \theta_t \]

or

\[ \theta_t = \frac{\alpha}{1 + \alpha} \theta_{t+1} + \frac{1}{1 + \alpha} \mu_t \]  
(9.42)

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 \( \bar{\mu} = \{\mu_t\}_{t=0}^{\infty} \) is square summable and we require that the sequence \( \bar{\theta} = \{\theta_t\}_{t=0}^{\infty} \) is square summable, the linear difference equation (9.42) can be solved forward to get:

\[ \theta_t = \frac{1}{1 + \alpha} \sum_{j=0}^{\infty} \left( \frac{\alpha}{1 + \alpha} \right)^j \mu_{t+j} \]  
(9.43)

**Insight:** In the spirit of Chang [Cha98], note that equations (9.41) and (9.43) show that \( \theta_t \) intermediates how choices of \( \mu_{t+j} \), \( j = 0, 1, \ldots \) impinge on time \( t \) real balances \( m_t - p_t = -\alpha \theta_t \)

We shall use this insight to help us simplify and analyze government policy problems 

That future rates of money creation influence earlier rates of inflation creates optimal government policy problems in which timing protocols matter 

We can rewrite the model as:

\[ \begin{bmatrix} 1 \\ \theta_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1+\alpha}{\alpha} \end{bmatrix} \begin{bmatrix} 1 \\ \theta_t \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{1}{\alpha} \end{bmatrix} \mu_t \]

or
\[ x_{t+1} = Ax_t + B\mu_t \] (9.44)

We write the model in the state-space form (9.44) even though \( \theta_0 \) is to be determined and so is not an initial condition as it ordinarily would be in the state-space model described in *Linear Quadratic Control*.

We write the model in the form (9.44) because we want to apply an approach described in *Stackelberg problems*.

Assume that a representative households utility of real balances at time \( t \) is:

\[ U(m_t - p_t) = a_0 + a_1(m_t - p_t) - \frac{a_2}{2}(m_t - p_t)^2, \quad a_0 > 0, a_1 > 0, a_2 > 0 \] (9.45)

The bliss level of real balances is then \( \frac{a_1}{a_2} \).

The money demand function (9.41) and the utility function (9.45) imply that the bliss level of real balances is attained when:

\[ \theta_t = \theta^* = -\frac{a_1}{a_2} \]

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 mans Friedman rule that attains Milton Friedmans *optimal quantity of money*).

Via equation (9.43), a government plan \( \tilde{\mu} = \{\mu_t\}_{t=0}^{\infty} \) leads to an equilibrium sequence of inflation outcomes \( \tilde{\theta} = \{\theta_t\}_{t=0}^{\infty} \).

We assume that social costs \( \frac{c}{2}\mu_t^2 \) are incurred at \( t \) when the government changes the stock of nominal money balances at rate \( \mu_t \).

Therefore, the one-period welfare function of a benevolent government is:

\[ -s(\theta_t, \mu_t) \equiv -r(x_t, \mu_t) = \begin{bmatrix} 1 \\ \theta_t \end{bmatrix}^T \begin{bmatrix} 1 & -\frac{a_0}{2} & -\frac{a_1}{a_2} \\ -\frac{a_0}{2} & -\frac{a_1}{2a_2} & \frac{c}{2} \end{bmatrix} \begin{bmatrix} 1 \\ \theta_t \end{bmatrix} - \frac{c}{2}\mu_t^2 = -x'R_{xt} - Q\mu_t^2 \] (9.46)

Household welfare is summarized by:

\[ v_0 = -\sum_{t=0}^{\infty} \beta^t r(x_t, \mu_t) = -\sum_{t=0}^{\infty} \beta^t s(\theta_t, \mu_t) \] (9.47)

We can represent \( v_0 \) recursively via

\[ v_t = s(\theta_t, \mu_t) + \beta v_{t+1} \] (9.48)

1116 Chapter 9. Dynamic Programming Squared
9.2.3 Structure

The following structure is induced by private agents behavior as summarized by the demand function for money \( (9.41) \) that leads to equation \( (9.43) \) that tells how future settings of \( \mu \) affect the current value of \( \theta \).

Equation \( (9.43) \) 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 \( \vec{\mu} \) is an arbitrary exogenous policy.

To make \( \vec{\mu} \) endogenous, we require a theory of government decisions.

9.2.4 Intertemporal Influences

Criterion function \( (9.47) \) and constraint system \( (9.44) \) exhibit the following structure:

- Setting \( \mu_t \neq 0 \) imposes costs \( \frac{\gamma}{2} \mu_t^2 \) at time \( t \) and at no other times; but
- The money growth rate \( \mu_t \) affects the representative households one-period utilities at all dates \( s = 0, 1, \ldots, t \).

That settings of \( \mu \) at one date affect household utilities at earlier dates sets the stage for the emergence of a time-inconsistent optimal government plan under a Ramsey (also called a Stackelberg) timing protocol.

9.2.5 Four Models of Government Policy

We consider four models of policy makers that differ in

- what a policy maker is allowed to choose, either a sequence \( \vec{\mu} \) or just a single period \( \mu_t \)
- when a policy maker chooses, either at time 0 or at times \( t \geq 0 \)
- what a policy maker assumes about how its choice of \( \mu_t \) affects private agents expectations about earlier and later inflation rates

In two of our models, there is a single policy maker that chooses a sequence \( \{\mu_t\}_{t=0}^{\infty} \) once and for all, taking into account how \( \mu_t \)'s at each date affect earlier components of household utilities.

- these two models thus employ a Ramsey or Stackelberg timing protocol

In two other models, there is a sequence of policy makers, each of whom sets \( \mu_t \) at one \( t \) only.

- Each such policy maker ignores effects that its choice of \( \mu_t \) has on household one-period utilities at dates \( s = 0, 1, \ldots, t - 1 \)

9.2. Ramsey plans, Time Inconsistency, Sustainable Plans 1117
The four models differ with respect to timing protocols, constraints on government policy, and government policy makers 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 policy makers chooses \( \mu_t \) for \( t = 0, 1, 2, \ldots \)
  - a time \( t \) policy maker chooses \( \mu_t \) only and forecasts that future government decisions are unaffected by its choice
- A sequence of separate policy makers chooses \( \mu_t \) for \( t = 0, 1, 2, \ldots \)
  - a time \( t \) policy maker chooses only \( \mu_t \) but believes that its choice of \( \mu_t \) shapes private agents beliefs about future rates of money creation and inflation, and through them, future government actions

### 9.2.6 A Ramsey Planner

First we consider a Ramsey planner that chooses \( \{\mu_t, \theta_t\}_{t=0}^{\infty} \) to maximize (9.47) subject to the law of motion (9.44)

We can split this problem into two stages, as in Stackelberg problems and [LS18] Chapter 19

In the first stage, we take the initial inflation rate \( \theta_0 \) as given, and then solve the resulting LQ dynamic programming problem

In the second stage, we maximize over the initial inflation rate \( \theta_0 \)

Define a feasible set of \((\vec{x}_1, \vec{\mu}_0)\) sequences:

\[
\Omega(x_0) = \{(\vec{x}_1, \vec{\mu}_0) : x_{t+1} = Ax_t + B\mu_t, \forall t \geq 0\}
\]

#### Subproblem 1

The value function

\[
J(x_0) = \max_{(\vec{x}_1, \vec{\mu}_0) \in \Omega(x_0)} \sum_{t=0}^{\infty} \beta^t r(x_t, \mu_t)
\]

satisfies the Bellman equation

\[
J(x) = \max_{\mu, x'} \{-r(x, \mu) + \beta J(x')\}
\]

subject to:

\[
x' = Ax + B\mu
\]
As in *Stackelberg problems*, we map this problem into a linear-quadratic control problem and then carefully use the optimal value function associated with it.

Guessing that \( J(x) = -x'Px \) and substituting into the Bellman equation gives rise to the algebraic matrix Riccati equation:

\[
P = R + \beta A'PA - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA
\]

and the optimal decision rule

\[
\mu_t = -Fx_t
\]

where

\[
F = \beta(Q + \beta B'PB)^{-1}B'PA
\]

The QuantEcon `LQ` class solves for \( F \) and \( P \) given inputs \( Q, R, A, B \), and \( \beta \).

**Subproblem 2**

The value of the Ramsey problem is

\[
V = \max_{x_0} J(x_0)
\]

The value function

\[
J(x_0) = -\begin{bmatrix} 1 & \theta_0 \end{bmatrix} \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix} \begin{bmatrix} 1 \\ \theta_0 \end{bmatrix} = -P_{11} - 2P_{21}\theta_0 - P_{22}\theta_0^2
\]

Maximizing this with respect to \( \theta_0 \) yields the FOC:

\[
-2P_{21} - 2P_{22}\theta_0 = 0
\]

which implies

\[
\theta_0^* = -\frac{P_{21}}{P_{22}}
\]

**Representation of Ramsey Plan**

The preceding calculations indicate that we can represent a Ramsey plan \( \tilde{\mu} \) recursively with the following system created in the spirit of Chang [Cha98]:

\[
\begin{align*}
\theta_0 &= \theta_0^* \\
\mu_t &= b_0 + b_1 \theta_t \\
\theta_{t+1} &= d_0 + d_1 \theta_t
\end{align*}
\]

To interpret this system, think of the sequence \( \{\theta_t\}_{t=0}^\infty \) as a sequence of synthetic promised inflation rates that are just tools for generating a sequence \( \tilde{\mu} \) of money growth rates that are to be substituted into equation (9.43) to form actual rates of inflation.
It can be verified that if we substitute a plan \( \vec{\tilde{\mu}} = \{\mu_t\}_{t=0}^{\infty} \) that satisfies these equations into equation (9.43), we obtain the same sequence \( \vec{\theta} \) generated by system (9.49).

Thus, our construction of a Ramsey plan guarantees that promised inflation equals actual inflation.

The inflation rate \( \theta_t \) that appears in system (9.49) and equation (9.43) plays three roles simultaneously:

- In equation (9.43), \( \theta_t \) is the actual rate of inflation between \( t \) and \( t + 1 \).
- In equation (9.42) and (9.43), \( \theta_t \) is also the public’s expected rate of inflation between \( t \) and \( t + 1 \).
- In system (9.49), \( \theta_t \) is a promised rate of inflation chosen by the Ramsey planner at time 0.

**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.

**9.2.7 A Constrained-to-a-Constant-\( \mu \) 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 government to choose different settings of its instrument at different moments, we now assume that at time 0, a government chooses a constant sequence \( \mu_t = \vec{\tilde{\mu}} \) for all \( t \geq 0 \) to maximize

\[
U(-\alpha \vec{\mu}) - \frac{c}{2} \vec{\mu}^2
\]

Here we have imposed the perfect foresight outcome that \( \theta_t = \vec{\tilde{\mu}} \) when the government chooses a constant \( \mu \) for all \( t \geq 0 \).

With the quadratic form (9.45) for the utility function \( U \), the maximizing \( \vec{\tilde{\mu}} \) is

\[
\vec{\tilde{\mu}} = -\frac{\alpha a_1}{\alpha^2 a_2 + c}
\]

**Summary**: We have introduced the constrained-to-a-constant-\( \mu \) government in order to highlight the role of time-variation of \( \mu_t \) in generating time inconsistency of a Ramsey plan.
9.2.8 Markov Perfect Governments

We now change the timing protocol by considering a government that chooses \( t \) and expects all future governments to set \( t + j = \tilde{\mu} \)

This assumption mirrors an assumption made in a different setting Markov Perfect Equilibrium

Further, the government at \( t \) believes that \( \tilde{\mu} \) is unaffected by its choice of \( \mu_t \)

The time \( t \) rate of inflation is then:

\[
\theta_t = \frac{\alpha}{1 + \alpha} \tilde{\mu} + \frac{1}{1 + \alpha} \mu_t
\]

The time \( t \) government policy maker then chooses \( \mu_t \) to maximize:

\[
W = U(-\alpha \theta_t) - \frac{c}{2} \mu_t^2 + \beta V(\tilde{\mu})
\]

where \( V(\tilde{\mu}) \) is the time 0 value \( v_0 \) of recursion (9.48) under a money supply growth rate that is forever constant at \( \tilde{\mu} \)

Substituting for \( U \) and \( \theta_t \) gives:

\[
W = a_0 + a_1 \left( -\frac{\alpha^2}{1 + \alpha} \tilde{\mu} - \frac{\alpha}{1 + \alpha} \mu_t \right) - a_2 \left( \frac{\alpha^2}{1 + \alpha} \tilde{\mu} - \frac{\alpha}{1 + \alpha} \mu_t \right)^2 - \frac{c}{2} \mu_t^2 + \beta V(\tilde{\mu})
\]

The first-order necessary condition for \( \mu_t \) is then:

\[
-\frac{\alpha}{1 + \alpha} a_1 - a_2 \left( -\frac{\alpha^2}{1 + \alpha} \tilde{\mu} - \frac{\alpha}{1 + \alpha} \mu_t \right) \left( -\frac{\alpha}{1 + \alpha} \mu_t \right) - c \mu_t = 0
\]

Rearranging we get:

\[
\mu_t = \frac{-a_1}{\alpha \frac{\alpha}{1 + \alpha} c + a_2} - \frac{\alpha^2 a_2}{\frac{\alpha}{1 + \alpha} c + \frac{\alpha}{1 + \alpha} a_2} \left( 1 + \alpha \right) \tilde{\mu}
\]

A Markov Perfect Equilibrium (MPE) outcome sets \( \mu_t = \tilde{\mu} \):

\[
\mu_t = \tilde{\mu} = \frac{-a_1}{\frac{\alpha}{1 + \alpha} c + \frac{\alpha}{1 + \alpha} a_2 + \frac{\alpha^2}{1 + \alpha} a_2}
\]

In light of results presented in the previous section, this can be simplified to:

\[
\tilde{\mu} = -\frac{\alpha a_1}{\alpha^2 a_2 + (1 + \alpha) c}
\]

9.2.9 Equilibrium Outcomes for Three Models of Government Policy Making

Below we compute sequences \( \{\theta_t, \mu_t\} \) under a Ramsey plan and compare these with the constant levels of \( \theta \) and \( \mu \) in a) a Markov Perfect Equilibrium, and b) a Ramsey plan in which the planner is restricted to choose \( \mu_t = \tilde{\mu} \) for all \( t \geq 0 \)

We denote the Ramsey sequence as \( \theta^R, \mu^R \) and the MPE values as \( \theta^{MPE}, \mu^{MPE} \)

The bliss level of inflation is denoted by \( \theta^* \)

First we will create a class \texttt{ChangLQ} that solves the models and stores their values
import numpy as np
from quantecon import LQ
import matplotlib.pyplot as plt

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))))
\[
\mu_{\text{series}} = \text{np.zeros}(T)
\]
\[
J_{\text{series}} = \text{np.zeros}(T)
\]
\[
\theta_{\text{series}}[1, 0] = \text{self.}\theta_{\text{R}}
\]
\[
\mu_{\text{series}}[0] = -\text{self.F.dot}(\theta_{\text{series}}[:, 0])
\]
\[
J_{\text{series}}[0] = -\theta_{\text{series}}[:, 0] \bowtie \text{self.P} \bowtie \theta_{\text{series}}[:, 0].T
\]

for i in range(1, T):
    \[
    \theta_{\text{series}}[:, i] = (A - B @ self.F) @ \theta_{\text{series}}[:, i-1]
    \]
    \[
    \mu_{\text{series}}[i] = -self.F @ \theta_{\text{series}}[:, i]
    \]
    \[
    J_{\text{series}}[i] = -\theta_{\text{series}}[:, i] \bowtie \text{self.P} \bowtie \theta_{\text{series}}[:, i].T
    \]

self.J_{\text{series}} = J_{\text{series}}
self.\mu_{\text{series}} = \mu_{\text{series}}
self.\theta_{\text{series}} = \theta_{\text{series}}

# Find range of \( \theta \) in Ramsey plan
\[
\theta_{\text{LB}} = \min(\theta_{\text{series}}[1, :])
\]
\[
\theta_{\text{UB}} = \min(\theta_{\text{LB}}, \text{self.}\theta_{\text{B}})
\]
\[
\theta_{\text{UB}} = \max(\theta_{\text{series}}[1, :])
\]
\[
\theta_{\text{UB}} = \max(\theta_{\text{UB}}, \text{self.}\theta_{\text{MPE}})
\]
\[
\theta_{\text{range}} = \theta_{\text{UB}} - \theta_{\text{LB}}
\]

self.\theta_{\text{LB}} = \theta_{\text{LB}} - 0.05 * \theta_{\text{range}}
self.\theta_{\text{UB}} = \theta_{\text{UB}} + 0.05 * \theta_{\text{range}}
self.\theta_{\text{range}} = \theta_{\text{range}}

# Find value function and policy functions over range of \( \theta \)
\[
\theta_{\text{space}} = \text{np.linspace}(\text{self.}\theta_{\text{LB}}, \text{self.}\theta_{\text{UB}}, 200)
\]
\[
J_{\text{space}} = \text{np.zeros}(200)
\]
\[
\text{check}_{\text{space}} = \text{np.zeros}(200)
\]
\[
\mu_{\text{space}} = \text{np.zeros}(200)
\]
\[
\theta_{\text{prime}} = \text{np.zeros}(200)
\]

for i in range(200):
    \[
    J_{\text{space}}[i] = -\text{np.array}((i, \theta_{\text{space}}[i])) \bowtie \text{self.P} \bowtie \text{np.array}((1, \theta_{\text{space}}[i])))^T
    \]
    \[
    \mu_{\text{space}}[i] = -self.F @ \text{np.array}((1, \theta_{\text{space}}[i]))
    \]
    \[
    x_{\text{prime}} = (A - B @ self.F) @ \text{np.array}((1, \theta_{\text{space}}[i]))
    \]
    \[
    \theta_{\text{prime}}[i] = x_{\text{prime}}[1]
    \]
    \[
    \text{check}_{\text{space}}[i] = (a0 + a1 * (-\alpha * \theta_{\text{space}}[i]) - a2/2 * (-\alpha * \theta_{\text{space}}[i])**2 - c/2 * \theta_{\text{space}}[i]**2) / (1 - self.\beta)
    \]

J_{\text{LB}} = \min(J_{\text{space}})
J_{\text{UB}} = \max(J_{\text{space}})
J_{\text{range}} = J_{\text{UB}} - J_{\text{LB}}

self.J_{\text{LB}} = J_{\text{LB}} - 0.05 * J_{\text{range}}
self.J_{\text{UB}} = J_{\text{UB}} + 0.05 * J_{\text{range}}
self.J_{\text{range}} = J_{\text{range}}
self.J_{\text{space}} = J_{\text{space}}
self.\theta_{\text{space}} = \theta_{\text{space}}
self.\mu_{\text{space}} = \mu_{\text{space}}
self.\theta_{\text{prime}} = \theta_{\text{prime}}
self.check_{\text{space}} = check_{\text{space}}

We will create an instance of ChangLQ with the following parameters

9.2. Ramsey plans, Time Inconsistency, Sustainable Plans

1123
The following code generates a figure that plots the value function from the Ramsey Planners problem, which is maximized at $\theta^R_0$.

The figure also shows the limiting value $\theta^R_\infty$ to which the inflation rate $\theta_t$ converges under the Ramsey plan and compares it to the MPE value and the bliss value.

```python
def plot_value_function(clq):
    """
    Method to plot the value function over the relevant range of $\theta$

    Here clq is an instance of ChangLQ
    """
    fig, ax = plt.subplots()
    ax.set_xlim([clq.theta_LB, clq.theta_UB])
    ax.set_ylim([clq.J_LB, clq.J_UB])

    # Plot value function
    ax.plot(clq.theta_space, clq.J_space, lw=2)
    plt.xlabel(r'$\theta$', fontsize=18)
    plt.ylabel(r'$J(\theta)$', fontsize=18)

    t1 = clq.theta_space[np.argmax(clq.J_space)]
    tR = clq.theta_series[1, -1]
    _points = [t1, tR, clq.theta_B, clq.theta_MPE]
    labels = [r'$\theta_0^R$', r'$\theta_\infty^R$', r'$\theta^*$', r'$\theta^{MPE}$']

    # Add points for $\theta$s
    for _theta, label in zip(_points, labels):
        ax.scatter(_theta, clq.J_LB + 0.02 * clq.J_range, 60, 'black', 'v')
        ax.annotate(label,
                    xy=(_theta, clq.J_LB + 0.01 * clq.J_range),
                    xytext=(_theta - 0.01 * clq.theta_range,
                             clq.J_LB + 0.08 * clq.J_range),
                    fontsize=18)

    plt.tight_layout()
    plt.show()
```

```
clq = ChangLQ(a=1, a0=1, a1=0.5, a2=3, c=2)
cdq

0.84648172489061413
```

The following code generates a figure that plots the value function from the Ramsey Planners problem, which is maximized at $\theta^R_0$.

The figure also shows the limiting value $\theta^R_\infty$ to which the inflation rate $\theta_t$ converges under the Ramsey plan and compares it to the MPE value and the bliss value.

```python
def plot_value_function(clq):
    """
    Method to plot the value function over the relevant range of $\theta$

    Here clq is an instance of ChangLQ
    """
    fig, ax = plt.subplots()
    ax.set_xlim([clq.theta_LB, clq.theta_UB])
    ax.set_ylim([clq.J_LB, clq.J_UB])

    # Plot value function
    ax.plot(clq.theta_space, clq.J_space, lw=2)
    plt.xlabel(r'$\theta$', fontsize=18)
    plt.ylabel(r'$J(\theta)$', fontsize=18)

    t1 = clq.theta_space[np.argmax(clq.J_space)]
    tR = clq.theta_series[1, -1]
    _points = [t1, tR, clq.theta_B, clq.theta_MPE]
    labels = [r'$\theta_0^R$', r'$\theta_\infty^R$', r'$\theta^*$', r'$\theta^{MPE}$']

    # Add points for $\theta$s
    for _theta, label in zip(_points, labels):
        ax.scatter(_theta, clq.J_LB + 0.02 * clq.J_range, 60, 'black', 'v')
        ax.annotate(label,
                    xy=(_theta, clq.J_LB + 0.01 * clq.J_range),
                    xytext=(_theta - 0.01 * clq.theta_range,
                             clq.J_LB + 0.08 * clq.J_range),
                    fontsize=18)

    plt.tight_layout()
    plt.show()
```

```
clq = ChangLQ(a=1, a0=1, a1=0.5, a2=3, c=2)
cdq

0.84648172489061413
```
The next code generates a figure that plots the value function from the Ramsey Planners problem as well as that for a Ramsey planner that must choose a constant $\mu$ (that in turn equals an implied constant $\theta$):

```python
def compare_ramsey_check(clq):
    
    """
    Method to compare values of Ramsey and Check
    """
    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._LB, clq._UB])
    ax.set_ylim([check_LB, check_UB])
    ax.plot(clq._space, clq.J_space, lw=2, label=r'$J(\theta)$')
    plt.xlabel(r'$\theta$', fontsize=18)
    ax.plot(clq._space, clq.check_space,
            lw=2, label=r'$V^{\check{\theta}}$')
    plt.legend(fontsize=14, loc='upper left')

    $\theta$ points = [clq._space[np.argmax(clq.J_space)],
                     clq._mu_check]
    labels = [r'$\theta^R_0$', r'$\theta^R_\infty$']
```

9.2. Ramsey plans, Time Inconsistency, Sustainable Plans
The next code generates figures that plot the policy functions for a continuation Ramsey planner:

```python
for θ, label in zip(θ_points, labels):
    ax.scatter(θ, check_LB + 0.02 * check_range, 60, 'k', 'v')
    ax.annotate(label,
        xy=(θ, check_LB + 0.01 * check_range),
        xytext=(θ - 0.02 * check_range, check_LB + 0.08 * check_range),
        fontsize=18)
plt.tight_layout()
plt.show()

compare_ramsey_check(clq)
```

The next code generates figures that plot the policy functions for a continuation Ramsey planner:

The left figure shows the choice of $\theta'$ chosen by a continuation Ramsey planner who inherits $\theta$

The right figure plots a continuation Ramsey planners choice of $\mu$ as a function of an inherited $\theta$

```python
def plot_policy_functions(clq):
    
    Method to plot the policy functions over the relevant range of $\theta$

    Here clq is an instance of ChangLQ

    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    labels = ['$\theta_0^R$', '$\theta_\infty^R$']
```

1126 Chapter 9. Dynamic Programming Squared
plot_policy_functions(clq)
The following code generates a figure that plots sequences of $\mu$ and $\theta$ in the Ramsey plan and compares these to the constant levels in a MPE and in a Ramsey plan with a government restricted to set $\mu_t$ to a constant for all $t$.

```python
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.\_series[1, 0:T], clq.\_series[0:T]]
    MPEs = [clq.\_MPE, clq.\_MPE]
    labels = [r"\theta", r"\mu"]
    axes[0].hlines(clq.\_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"\mu")
        ax.hlines(MPE, 0, T-1, 'orange', label=r"" + label + r"\mu")
        ax.hlines(clq.\_check, 0, T, 'g', label=r"" + label + r"\check")
        ax.set_xlabel(r"t", fontsize=16)
        ax.set_ylabel(r"t", fontsize=16)
        ax.legend(loc='upper right')
    plt.tight_layout()
    plt.show()

plot_ramsey_MPE(clq)
```
Time Inconsistency of Ramsey Plan

The variation over time in $\hat{\mu}$ 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 (9.43)
- A government authority offered the opportunity to ignore effects on past utilities and to reoptimize at date $t \geq 1$ would, if allowed, want to deviate from a Ramsey plan

Note: A modified Ramsey plan constructed under the restriction that $\mu_t$ must be constant over time is time consistent (see $\hat{\mu}$ and $\hat{\theta}$ in the above graphs)

Meaning of Time Inconsistency

In settings in which governments 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 policy makers 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 policy makers 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

Ramsey Plan Strikes Back

Research by Abreu [Abr88], Chari and Kehoe [CK90] [Sto89], and [Sto91] discovered conditions under which a Ramsey plan can be rescued from the complaint that it is not credible
They accomplished this by expanding the description of a plan to include expectations about adverse consequences of deviating from it that can serve to deter deviations.

We turn to such theories of sustainable plans next.

### 9.2.10 A Fourth Model of Government Decision Making

This is a model in which

- The government chooses \( \{ \mu_t \}_{t=0}^{\infty} \) not once and for all at \( t = 0 \) but sequentially.
- Private agents forecasts of \( \{ \mu_{t+j+1}, \theta_{t+j+1} \}_{j=0}^{\infty} \) respond to whether the government at \( t \) confirms or disappoints their forecasts of \( \mu_t \) brought into period \( t \) from period \( t-1 \).
- The government at each time \( t \) understands how private agents forecasts will respond to its choice of \( \mu_t \).
- At each \( t \), the government chooses \( \mu_t \) to maximize a continuation discounted utility of a representative household.

### A Theory of Government Decision Making

\( \bar{\mu} \) 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 \( \mu_t = \bar{\mu}_t \), and more generally that it will set \( \mu_{t+j} = \bar{\mu}_{t+j} \) for all \( j \geq 0 \).
- Those forecasts determine a \( \theta_t = \bar{\theta}_t \) and an associated log of real balances \( m_t - p_t = -\alpha \bar{\theta}_t \) at \( t \).
- Given those expectations and the associated \( \theta_t \), at \( t \) a government is free to set \( \mu_t \in \mathbb{R} \).
- If the government at \( t \) confirms private agents expectations by setting \( \mu_t = \bar{\mu}_t \) at time \( t \), private agents expect the continuation government policy \( \{ \bar{\mu}_{t+j+1} \}_{j=0}^{\infty} \) and therefore bring expectation \( \bar{\theta}_{t+1} \) into period \( t+1 \).
- If the government disappoints private agents by setting \( \mu_t \neq \bar{\mu}_t \), private agents expect \( \{ \mu^A_j \}_{j=0}^{\infty} \) as the continuation policy for \( t+1 \), i.e., \( \{ \mu_{t+j+1} \} = \{ \mu^A_j \}_{j=0}^{\infty} \) and therefore expect \( \theta^A_0 \) for \( t+1 \). Here \( \mu^A = \{ \mu^A_j \}_{j=0}^{\infty} \) is an alternative government plan to be described below.

### Temptation to Deviate from Plan

The government's one-period return function \( s(\theta, \mu) \) described in equation (9.46) above has the property that for all \( \theta \)

\[
s(\theta, 0) \geq s(\theta, \mu)
\]

This inequality implies that whenever the policy calls for the government to set \( \mu \neq 0 \), the government could raise its one-period return by setting \( \mu = 0 \).
Disappointing private sector expectations in that way would increase the governments current payoff but would have adverse consequences for subsequent government payoffs because the private sector would alter its expectations about future settings of $\mu_t$.

The temporary gain constitutes the governments temptation to deviate from a plan.

If the government at $t$ is to resist the temptation to raise its current payoff, it is only because it forecasts adverse consequences that its setting of $\mu_t$ would bring for subsequent government payoffs via alterations in the private sectors expectations.

### 9.2.11 Sustainable or Credible plan

We call a plan $\vec{\mu}$ sustainable or credible if at each $t \geq 0$ the government chooses to confirm private agents prior expectation of its setting for $\mu_t$.

The government will choose to confirm prior expectations if the long-term loss from disappointing private sector expectations – coming from the governments 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 $t \geq 0$ will act to maximize the continuation discounted utilities that describe those governments purposes.

This aspect of the theory means that credible plans 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 seems 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.

### Abreus Self-Enforcing Plan

A plan $\vec{\mu}^A$ is said to be self-enforcing if

- the consequence of disappointing private agents expectations at time $j$ is to restart the plan at time $j + 1$
- that consequence is sufficiently adverse that it deters all deviations from the plan

More precisely, a government plan $\vec{\mu}^A$ is self-enforcing if
\[ v_j^A = s(\theta_j^A, \mu_j^A) + \beta v_{j+1}^A \]
\[ \geq s(\theta_j^A, 0) + \beta v_0^A \equiv v_j^{A,D}, \quad j \geq 0 \] (9.50)

(Here it is useful to recall that setting \( \mu = 0 \) is the maximizing choice for the governments one-period return function)

The first line tells the consequences of confirming private agents expectations, while the second line tells the consequences of disappointing private agents expectations

A consequence of the definition is that a self-enforcing plan is credible

Self-enforcing plans can be used to construct other credible plans, ones with better values

A sufficient condition for a plan \( \bar{\mu} \) to be **credible** or **sustainable** is that

\[ \bar{v}_j = s(\bar{\theta}_j, \mu_j) + \beta \bar{v}_{j+1} \]
\[ \geq s(\bar{\theta}_j, 0) + \beta v_0^A \quad \forall j \geq 0 \]

A 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 when the governments choice at time \( t \) fails to confirm private agents expectation

We shall use a construction featured in \([Abr88]\) to construct a self-enforcing plan with low time 0 value

**Abreu Carrot-Stick Plan**

\([Abr88]\) invented a way to create a self-enforcing plan with low initial value.

Imitating his idea, we can construct a self-enforcing plan \( \bar{\mu} \) with a low time 0 value to the government by insisting that the government set \( \mu_t \) to a value yielding low one-period utilities to the household for a long time, after which the government forever does things yielding high one-period utilities

- low one-period utilities early are a **stick**
- high one-period utilities later are a **carrot**

Consider a plan \( \bar{\mu}^A \) in which the government 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 \( \{\mu_t^A\}_{t=0}^{T_A} \)

The sequence of inflation rates implied by this plan, \( \{\theta_t^A\}_{t=0}^{T_A} \), can be calculated using:

\[ \theta_t^A = \frac{1}{1 + \alpha} \sum_{j=0}^{T_A} \left( \frac{\alpha}{1 + \alpha} \right)^j \mu_{t+j}^A \]

The value of \( \{\theta_t^A, \mu_t^A\}_{t=0}^{T_A} \) is

\[ v_0^A = \sum_{t=0}^{T_A-1} \beta^t s(\theta_t^A, \mu_t^A) + \beta^{T_A} J(\theta_0^R) \]
Example of Self-Enforcing Plan

The following example implements Abreus idea.

The government sets $\mu_t^A = 0.1$ for $t = 0, 1, \ldots, 9$ and then starts the Ramsey plan.

We have computed outcomes for this plan.

For this plan, we plot the $\theta^A$, $\mu^A$ sequences as well as the implied $v^A$ sequence.

Notice that because the government sets money supply growth high for 10 periods, inflation starts high.

Inflation gradually slows down immediately because people immediately expect the government to lower the money growth rate after period 10.

From the 10th period onwards, the inflation rate $\theta_t^A$ associated with this Abreu plan starts the Ramsey plan from its beginning, i.e., $\theta_{t+10} = \theta_t^R \forall t \geq 0$.

```python
def abreu_plan(clq, T=1000, T_A=10, _bar=0.1, T_Plot=20):
    # Append Ramsey $\mu$ series to stick $\mu$ series
    clq._A = np.append(np.ones(T_A) * _bar, clq._series[-T_A:])

    # Calculate implied stick $\theta$ series
    clq._A = np.zeros(T)
    discount = np.zeros(T)
    for t in range(T):
        discount[t] = (clq._A[t] / (1 + clq._A[t]))**t

    # Calculate utility of stick plan
    U_A = np.zeros(T)
    for t in range(T):

    # Make sure Abreu plan is self-enforcing
    clq._V_dev = np.zeros(T_Plot)
    for t in range(T_Plot):

    fig, axes = plt.subplots(3, 1, figsize=(8, 12))
    axes[2].plot(clq._V_dev[0:T_Plot], label=r"$V^A_t$", c="orange")
    plots = [clq._A, clq._A, clq._A]
    labels = [r"$\theta_t^A$", r"$\mu_t^A$", r"$V^A_t$"]
```

9.2. Ramsey plans, Time Inconsistency, Sustainable Plans 1133
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)
9.2. Ramsey plans, Time Inconsistency, Sustainable Plans
To confirm that the plan $\vec{v}^A$ is self-enforcing, the right panel plots an object that we call $V^A_{t,D}$, defined in the second line of equation (9.50) above:

$V^A_{t,D}$ is the value at $t$ of deviating from the self-enforcing plan $\vec{v}^A$ by setting $\mu_t = 0$ and then restarting the plan at $v_0^A$ at $t + 1$.

Notice that $v_t^A > v_t^{A,D}$.

This confirms that $\vec{v}^A$ is a self-enforcing plan.

We can also verify the inequalities required for $\vec{v}^A$ to be self-confirming numerically as follows:

```python
np.all(clq.V_A[0:20] > clq.V_dev[0:20])
```

True

Given that plan $\vec{v}^A$ is self-enforcing, we can check that the Ramsey plan $\vec{v}^R$ is sustainable by verifying that:

$$v_t^R \geq s(\theta^R_t, 0) + \beta v_0^A, \quad \forall t \geq 0$$

```python
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.a0 + clq.a1 * (-clq.theta_series[1, t]) - clq.a2 / 2 * (-clq.theta_series[1, t])**2) + clq.beta * clq.V_A[0]
    return np.all(clq.J_series > R_dev)
check_ramsey(clq)
```

True

**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:

$$\begin{align*}
\hat{\mu}_t &= \nu_\mu(v_t) \\
\theta_t &= \nu_\theta(v_t) \\
v_{t+1} &= \nu_v(v_t, \hat{\mu}_t)
\end{align*}$$

(9.51)

In addition to these equations, we need an initial value $v_0$ to characterize a sustainable plan.

The first equation of (9.51) tells the recommended value of $\hat{\mu}_t$ as a function of the promised value $v_t$.

The second equation of (9.51) tells the inflation rate as a function of $v_t$. 

Chapter 9. Dynamic Programming Squared
The third equation of (9.51) updates the continuation value in a way that depends on whether the government at \( t \) confirms private agents expectations by setting \( \mu_t \) equal to the recommended value \( \hat{\mu}_t \), or whether it disappoints those expectations.

### 9.2.12 Comparison of Equilibrium Values

We have computed plans for:
- an ordinary (unrestricted) Ramsey planner who chooses a sequence \( \{\mu_t\}^\infty_{t=0} \) at time 0
- a Ramsey planner restricted to choose a constant \( \mu \) for all \( t \geq 0 \)
- a Markov perfect sequence of governments

Below we compare equilibrium time zero values for these three:

We confirm that the value delivered by the unrestricted Ramsey planner exceeds the value delivered by the restricted Ramsey planner which in turn exceeds the value delivered by the Markov perfect sequence of governments.

<table>
<thead>
<tr>
<th>Plan Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restrict</td>
<td>6.67918822960449</td>
</tr>
<tr>
<td>Restricted</td>
<td>6.6767295246748981</td>
</tr>
<tr>
<td>Markov Perfect</td>
<td>6.6634358869951074</td>
</tr>
</tbody>
</table>

We have also computed **sustainable plans** for a government or sequence of governments that chooses 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

### 9.2.13 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 (9.41) expresses how \( \theta_t \) depends on \( \mu_t \) and \( \theta_{t+1} \)
• equation (9.44) expresses how value $v_t$ depends on $(\mu_t, \theta_t)$ and $v_{t+1}$

A value $\theta$ from one Bellman equation appears as an argument of a second Bellman equation for another value $v$

9.3 Optimal Taxation in an LQ Economy

Contents

• Optimal Taxation in an LQ Economy
  – Overview
  – The Ramsey Problem
  – Implementation
  – Examples
  – Exercises
  – Solutions

9.3.1 Overview

In this lecture we study optimal fiscal policy in a linear quadratic setting

We slightly modify a well-known model of Robert Lucas and Nancy Stokey [LS83] so that convenient formulas for solving linear-quadratic models can be applied to simplify the calculations

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 governments tax rate and borrowing plans as given

Maximum attainable utility for the household depends on the governments tax and borrowing plans

The Ramsey problem [Ram27] is to choose tax and borrowing plans that maximize the households welfare, taking the households optimizing behavior as given

There is a large number of competitive equilibria indexed by different government fiscal policies

The Ramsey planner chooses the best competitive equilibrium

We want to study the dynamics of tax rates, tax revenues, government debt under a Ramsey plan

Because the Lucas and Stokey model features state-contingent government debt, the government debt dynamics differ substantially from those in a model of Robert Barro [Bar79]
The treatment given here closely follows this manuscript, prepared by Thomas J. Sargent and Francois R. Velde.

We cover only the key features of the problem in this lecture, leaving you to refer to that source for additional results and intuition.

**Model Features**

- Linear quadratic (LQ) model
- Representative household
- Stochastic dynamic programming over an infinite horizon
- Distortionary taxation

**9.3.2 The Ramsey Problem**

We begin by outlining the key assumptions regarding technology, households and the government sector.

**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.

**Households**

Consider a representative household who chooses a path \( \{ \ell_t, c_t \} \) for labor and consumption to maximize

\[
-E \frac{1}{2} \sum_{t=0}^{\infty} \beta^t \left[ (c_t - b_t)^2 + \ell_t^2 \right]
\]  

subject to the budget constraint

\[
E \sum_{t=0}^{\infty} \beta^t p_t^0 \left[ d_t + (1 - \tau_t) \ell_t + s_t - c_t \right] = 0
\]  

Here

- \( \beta \) is a discount factor in \((0, 1)\)
- \( p_t^0 \) is a scaled Arrow-Debreu price at time 0 of history contingent goods at time \( t + j \)
• $b_t$ is a stochastic preference parameter
• $d_t$ is an endowment process
• $\tau_t$ is a flat tax rate on labor income
• $s_t$ is a promised time-$t$ coupon payment on debt issued by the government

The scaled Arrow-Debreu price $p^0_t$ is related to the unscaled Arrow-Debreu price as follows.

If we let $\pi^0_t(x^t)$ denote the probability (density) of a history $x^t = [x_t, x_{t-1}, \ldots, x_0]$ 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

$$\frac{\beta^t p^0_t}{\pi^0_t(x^t)}$$

Thus, our scaled Arrow-Debreu price is the ordinary Arrow-Debreu price multiplied by the discount factor $\beta^t$ and divided by an appropriate probability.

The budget constraint (9.53) requires that the present value of consumption be restricted to equal the present value of endowments, labor income and coupon payments on bond holdings.

**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.

**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 $x_{t+1} = A x_t + C w_{t+1}$ where $\{w_t\}$ is independent zero mean Gaussian with identify covariance matrix
Feasibility

The period-by-period feasibility restriction for this economy is

\[ c_t + g_t = d_t + \ell_t \]  \hspace{1cm} (9.54)

A labor-consumption process \( \{\ell_t, c_t\} \) is called feasible if (9.54) holds for all \( t \)

Government budget constraint

Where \( p^0_t \) is again a scaled Arrow-Debreu price, the time zero government budget constraint is

\[ \mathbb{E} \sum_{t=0}^{\infty} \beta^t p^0_t (s_t + g_t - \tau_t \ell_t) = 0 \]  \hspace{1cm} (9.55)

Equilibrium

An equilibrium is a feasible allocation \( \{\ell_t, c_t\} \), a sequence of prices \( \{p^0_t\} \), and a tax system \( \{\tau_t\} \) such that

1. The allocation \( \{\ell_t, c_t\} \) is optimal for the household given \( \{p^0_t\} \) and \( \{\tau_t\} \)
2. The governments budget constraint (9.55) is satisfied

The Ramsey problem is to choose the equilibrium \( \{\ell_t, c_t, \tau_t, p^0_t\} \) that maximizes the households welfare

If \( \{\ell_t, c_t, \tau_t, p^0_t\} \) 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 (9.55) in terms of exogenous variables and allocations
3. Maximize the households objective function (9.52) subject to the constraint constructed in step 2 and the feasibility constraint (9.54)

The solution to this maximization problem pins down all quantities of interest

Solution

Step one is to obtain the first-conditions for the households problem, taking taxes and prices as given

Letting \( \mu \) be the Lagrange multiplier on (9.53), the first-order conditions are \( p^0_t = (c_t - b_t)/\mu \) 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
\[ p_t^0 = \frac{b_t - c_t}{b_0 - c_0} \quad \text{and} \quad \tau_t = 1 - \frac{\ell_t}{b_t - c_t} \quad (9.56) \]

Substituting (9.56) into the governments budget constraint (9.55) yields

\[ \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left[ (b_t - c_t)(s_t + g_t - \ell_t) + \ell_t^2 \right] = 0 \quad (9.57) \]

The Ramsey problem now amounts to maximizing (9.52) subject to (9.57) and (9.54)

The associated Lagrangian is

\[ \mathcal{L} = \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left\{ -\frac{1}{2} [ (c_t - b_t)^2 + \ell_t^2 ] + \lambda \left[ (b_t - c_t)(\ell_t - s_t - g_t) - \ell_t^2 \right] + \mu_t [d_t + \ell_t - c_t - g_t] \right\} \quad (9.58) \]

The first order conditions associated with \( c_t \) and \( \ell_t \) are

\[-(c_t - b_t) + \lambda [ -\ell_t + (g_t + s_t) ] = \mu_t \]

and

\[ \ell_t - \lambda [(b_t - c_t) - 2\ell_t] = \mu_t \]

Combining these last two equalities with (9.54) and working through the algebra, one can show that

\[ \ell_t = \bar{\ell}_t - \nu m_t \quad \text{and} \quad c_t = \bar{c}_t - \nu m_t \quad (9.59) \]

where

- \( \nu := \lambda/(1 + 2\lambda) \)
- \( \bar{\ell}_t := (b_t - d_t + g_t)/2 \)
- \( \bar{c}_t := (b_t + d_t - g_t)/2 \)
- \( m_t := (b_t - d_t - s_t)/2 \)

Apart from \( \nu \), all of these quantities are expressed in terms of exogenous variables

To solve for \( \nu \), we can use the governments budget constraint again

The term inside the brackets in (9.57) is \((b_t - c_t)(s_t + g_t) - (b_t - c_t)\ell_t + \ell_t^2\)

Using (9.59), 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 (9.57), we get
\[
\mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t (b_t - \bar{c}_t)(g_t + s_t) \right\} + (\nu^2 - \nu) \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t 2m_t^2 \right\} = 0 \quad (9.60)
\]

Although it might not be clear yet, we are nearly there because:

- The two expectations terms in (9.60) can be solved for in terms of model primitives
- This in turn allows us to solve for the Lagrange multiplier \( \nu \)
- With \( \nu \) in hand, we can go back and solve for the allocations via (9.59)
- Once we have the allocations, prices and the tax system can be derived from (9.56)

**Computing the Quadratic Term**

Let's consider how to obtain the term \( \nu \) in (9.60)

If we can compute the two expected geometric sums

\[
b_0 := \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t (b_t - \bar{c}_t)(g_t + s_t) \right\} \quad \text{and} \quad a_0 := \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t 2m_t^2 \right\} \quad (9.61)
\]

then the problem reduces to solving

\[
b_0 + a_0(\nu^2 - \nu) = 0
\]

for \( \nu \)

Provided that \( 4b_0 < a_0 \), there is a unique solution \( \nu \in (0, 1/2) \), and a unique corresponding \( \lambda > 0 \)

Let's work out how to compute mathematical expectations in (9.61)

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 (9.61), 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

\[
q(x_0) = \mathbb{E} \sum_{t=0}^{\infty} \beta^t x_t' H x_t \quad (9.62)
\]

where \( H \) is a matrix conformable to \( x_t \) and \( x_t' \) is the transpose of column vector \( x_t \)

Suppose first that \( \{x_t\} \) is the Gaussian VAR described above

In this case, the formula for computing \( q(x_0) \) is known to be \( q(x_0) = x_0' Q x_0 + \nu \), 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.

**Finite state Markov case**

Next suppose that $\{x_t\}$ is the discrete Markov process described above.

Suppose further that each $x_t$ takes values in the state space $\{x^1, \ldots, x^N\} \subset \mathbb{R}^k$.

Let $h: \mathbb{R}^k \to \mathbb{R}$ be a given function, and suppose that we wish to evaluate

$$q(x_0) = \mathbb{E}\sum_{t=0}^{\infty} \beta^t h(x_t) \quad \text{given} \quad x_0 = x^j$$

For example, in the discussion above, $h(x_t) = x_t' H x_t$

It is legitimate to pass the expectation through the sum, leading to

$$q(x_0) = \sum_{t=0}^{\infty} \beta^t (P^t h)[j] \quad (9.63)$$

Here

• $P^t$ is the $t$-th power of the transition matrix $P$
• $h$ is, with some abuse of notation, the vector $(h(x^1), \ldots, h(x^N))$
• $(P^t h)[j]$ indicates the $j$-th element of $P^t h$

It can be show that (9.63) 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.

**Other Variables**

We are interested in tracking several other variables besides the ones described above.

To prepare the way for this, we define

$$p^t_{t+j} = \frac{b_{t+j} - c_{t+j}}{b_t - c_t}$$

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 market 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
\[ B_t := \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j p_{t+j}^t (\tau_{t+j} \ell_{t+j} - g_{t+j}) \quad (9.64) \]

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 (9.64) implies that

\[ B_t = (\tau_t \ell_t - g_t) + \mathbb{E}_t \sum_{j=1}^{\infty} p_{t+j}^t (\tau_{t+j} \ell_{t+j} - g_{t+j}) \]

and

\[ B_t = (\tau_t \ell_t - g_t) + \beta \mathbb{E}_t p_{t+1}^t B_{t+1} \quad (9.65) \]

Define

\[ R_t^{-1} := \mathbb{E}_t \beta^j p_{t+1}^t \quad (9.66) \]

\( R_t \) is the gross 1-period risk-free rate for loans between \( t \) and \( t+1 \).

**A Martingale**

We now want to study the following two objects, namely,

\[ \pi_{t+1} := B_{t+1} - R_t [B_t - (\tau_t \ell_t - g_t)] \]

and the cumulation of \( \pi_t \)

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

The term \( \pi_{t+1} \) is the difference between two quantities:

- \( B_{t+1} \), the value of government debt at the start of period \( t+1 \)
- \( R_t [B_t + g_t - \tau_t] \), which is what the government would have owed at the beginning of period \( t+1 \) if it had simply borrowed at the one-period risk-free rate rather than selling state-contingent securities
Thus, \( \pi_{t+1} \) is the excess payout on the actual portfolio of state contingent government debt relative to an alternative portfolio sufficient to finance \( B_t + g_t - \tau_t \ell_t \) and consisting entirely of risk-free one-period bonds. Use expressions (9.65) and (9.66) 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

\[
\pi_{t+1} = B_{t+1} - \tilde{E}_t B_{t+1} \quad (9.67)
\]

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 (9.67) that

\[
\tilde{E}_t \pi_{t+1} = \tilde{E}_t B_{t+1} - \tilde{E}_t B_{t+1} = 0
\]

which asserts that \( \{\pi_{t+1}\} \) is a martingale difference sequence under the distorted probability measure, and that \( \{\Pi_t\} \) is a martingale under the distorted probability measure.

In the tax-smoothing model of Robert Barro [Bar79], government debt is a random walk. In the current model, government debt \( \{B_t\} \) is not a random walk, but the excess payoff \( \{\Pi_t\} \) on it is

### 9.3.3 Implementation

The following code provides functions for

1. Solving for the Ramsey plan given a specification of the economy
2. Simulating the dynamics of the major variables

Description and clarifications are given below

```python
import sys
import numpy as np
from numpy import sqrt, eye, zeros, cumsum
from numpy.random import randn
import scipy.linalg
import matplotlib.pyplot as plt
from collections import namedtuple
from quantecon import nullspace, mc_sample_path, var_quadratic_sum

# == Set up a namedtuple to store data on the model economy == #
Economy = namedtuple('economy',
                     ('\'\beta\'',)  # 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


9.3. Optimal Taxation in an LQ Economy 1147
- Tax rate
- Revenue
- Govt debt
- Risk free gross return
- One-period risk-free interest rate
- Cumulative rate of return, adjusted
- Adjustment factor for

The corresponding values are flat numpy ndarrays.

```python
# == 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, nw = A.shape
    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 = l - 1 / (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

9.3. Optimal Taxation in an LQ Economy
1149
AF1 = (Sb - Sc) @ x[:, 1:]
AF2 = (Sb - Sc) @ A @ x[:, :T-1]
ξ = AF1 / AF2
ξ = ξ.flatten()

π = B[1:] - R[:T-1] @ A @ x[:, :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.pi, 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()  

Comments on the Code

The function var_quadratic_sum imported from quadsums is for computing the value of (9.62) 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

9.3. Optimal Taxation in an LQ Economy
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.

### 9.3.4 Examples

Let's look at two examples of usage.

#### 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:

```python
# == Parameters == #
\beta = 1 / 1.05
\rho, mg = .7, .35
A = eye(2)
A[0, :] = \rho, mg * (1-\rho)
C = np.zeros((2, 1))
C[0, 0] = np.sqrt(1 - \rho**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=\beta, Sg=Sg, Sd=Sd, Sb=Sb, Ss=Ss, discrete=False, proc=(A, C))
```

\( T = 50 \)

```python
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.

```python
gen_fig_2(path)
```

9.3. Optimal Taxation in an LQ Economy 1153
The Discrete Case

Our second example adopts a discrete Markov specification for the exogenous process

```python
# == Parameters ==
\beta = 1 / 1.05
P = np.array([[0.8, 0.2, 0.0],
              [0.0, 0.5, 0.5],
              [0.0, 0.0, 1.0]])
```

See the original manuscript for comments and interpretation
# == Possible states of the world == #
# Each column is a state of the world. The rows are [g d b s l]
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

9.3. Optimal Taxation in an LQ Economy
gen_fig_2(path)

See the original manuscript for comments and interpretation
### 9.3.5 Exercises

**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

### 9.3.6 Solutions

**Exercise 1**

```python
# == Parameters == #
\beta = 1 / 1.05
\rho, mg = .95, .35
A = np.array([[0, 0, 0, \rho, mg*(1-\rho)],
              [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 - \rho**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)
Sb = np.array((0, 0, 0, 0, 2.135)).reshape(1, 5)  # Chosen st. (Sg + Sd) + x0 = 1
Ss = np.array((0, 0, 0, 0, 0)).reshape(1, 5)

economy = Economy(\beta=\beta, Sg=Sg, Sd=Sd, Sb=Sb,
                   Ss=Ss, discrete=False, proc=(A, C))

T = 50
path = compute_paths(T, economy)
gen_fig_1(path)
```

---

9.3. Optimal Taxation in an LQ Economy

1157
gen_fig_2(path)
9.4 Optimal Taxation with State-Contingent Debt

Contents

- Optimal Taxation with State-Contingent Debt
  - Overview
  - A Competitive Equilibrium with Distorting Taxes
## 9.4.1 Overview

This lecture describes a celebrated model of optimal fiscal policy by Robert E. Lucas, Jr., and Nancy Stokey \([LS83]\)

The model revisits classic issues about how to pay for a war

Here a war means a more or less temporary surge in an exogenous government expenditure process

The model features

1. 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
2. a representative household that values consumption and leisure
3. a linear production function mapping labor into a single good
4. 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 an analysis within a linear-quadratic setting

## 9.4.2 A Competitive Equilibrium with Distorting Taxes

For \(t \geq 0\), a history \(s^t = [s_t, s_{t-1}, \ldots, s_0]\) of an exogenous state \(s_t\) has joint probability density \(\pi_t(s^t)\)

We begin by assuming that government purchases \(g_t(s^t)\) at time \(t \geq 0\) depend on \(s^t\)

Let \(c_t(s^t), \ell_t(s^t),\) and \(n_t(s^t)\) denote consumption, leisure, and labor supply, respectively, at history \(s^t\) and date \(t\)

A representative household is endowed with one unit of time that can be divided between leisure \(\ell_t\) and labor \(n_t\):

\[
    n_t(s^t) + \ell_t(s^t) = 1 \tag{9.68}
\]

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{9.69} 

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{9.70}

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

\[
g_t(s^t) = \tau_t(s^t)n_t(s^t) + \sum_{s_{t+1}} p_{t+1}(s_{t+1}|s^t)b_{t+1}(s_{t+1}|s^t) - b_t(s_t|s^{t-1}) \tag{9.71}
\]

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

\[
c_t(s^t) + \sum_{s_{t+1}} p_t(s_{t+1}|s^t)b_{t+1}(s_{t+1}|s^t) = \left[1 - \tau_t(s^t)\right]n_t(s^t) + b_t(s_t|s^{t-1}) \quad \forall t \geq 0. \tag{9.72}
\]

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 (9.69) 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 (9.70) subject to (9.72) and (9.68) 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 households optimization problem

• Given the allocation, government policy, and price system, the governments 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 (9.70)

### 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 $q^0_t(s^t)$ 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 $\{q^0_t(s^t)\}_{t=0}^{\infty}$ to Arrow securities prices $\{p_{t+1}(s_{t+1}|s^t)\}_{t=0}^{\infty}$:

$$
q^0_{t+1}(s^{t+1}) = p_{t+1}(s_{t+1}|s^t)q^0_t(s^t) \quad s.t. \quad q^0_0(s^0) = 1 \tag{9.73}
$$

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

### 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 households problem and solve them for $\{q^0_t(s^t), \tau_t(s^t)\}_{t=0}^{\infty}$ 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 households present-value budget constraint

   • This intertemporal constraint involves only the allocation and is regarded as an **implementability constraint**

3. Find the allocation that maximizes the utility of the representative household (9.70) subject to the feasibility constraints (9.68) and (9.69) and the implementability condition derived in step 2
This optimal allocation is called the **Ramsey allocation**

4. Use the Ramsey allocation together with the formulas from step 1 to find taxes and prices

### The Implementability Constraint

By sequential substitution of one one-period budget constraint (9.72) into another, we can obtain the households present-value budget constraint:

\[
\sum_{t=0}^{\infty} \sum_{s^t} q_t^0(s^t) c_t(s^t) = \sum_{t=0}^{\infty} \sum_{s^t} q_t^0(s^t)[1 - \tau_t(s^t)] n_t(s^t) + b_0
\]  
(9.74)

\{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 households optimization problem

First-order conditions for the households problem for \(\ell_t(s^t)\) and \(b_t(s_{t+1}|s^t)\), respectively, imply

\[
(1 - \tau_t(s^t)) = \frac{u_t(s^t)}{u_c(s^t)}
\]  
(9.75)

and

\[
p_{t+1}(s_{t+1}|s^t) = \beta \pi(s_{t+1}|s^t) \left( \frac{u_c(s^{t+1})}{u_c(s^t)} \right)
\]  
(9.76)

where \(\pi(s_{t+1}|s^t)\) is the probability distribution of \(s_{t+1}\) conditional on history \(s^t\)

Equation (9.76) implies that the Arrow-Debreu price system satisfies

\[
q_t^0(s^t) = \beta^t \pi_t(s^t) \frac{u_c(s^t)}{u_c(s^0)}
\]  
(9.77)

Using the first-order conditions (9.75) and (9.76) to eliminate taxes and prices from (9.74), we derive the **implementability condition**

\[
\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t)[u_c(s^t)c_t(s^t) - u_t(s^t)n_t(s^t)] - u_c(s^0)b_0 = 0.
\]  
(9.78)

The **Ramsey problem** is to choose a feasible allocation that maximizes

\[
\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t)u[c_t(s^t), 1 - n_t(s^t)]
\]  
(9.79)

subject to (9.78)
Solution Details

First define a pseudo utility function

\[ V[c_t(s^t), n_t(s^t), \Phi] = u[c_t(s^t), 1 - n_t(s^t)] + \Phi [u_c(s^t)c_t(s^t) - u_{\ell}(s^t)n_t(s^t)] \] (9.80)

where \( \Phi \) is a Lagrange multiplier on the implementability condition (9.74)

Next form the Lagrangian

\[ J = \sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) \left\{ V[c_t(s^t), n_t(s^t), \Phi] + \theta_t(s^t) \left[ n_t(s^t) - c_t(s^t) - g_t(s_t) \right] \right\} - \Phi u_c(0)b_0 \] (9.81)

where \( \{\theta_t(s^t); \forall s^t\}_{t \geq 0} \) is a sequence of Lagrange multipliers on the feasible conditions (9.69)

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

\[
\begin{align*}
c_t(s^t): & \quad (1 + \Phi)u_c(s^t) + \Phi [u_{cc}(s^t)c_t(s^t) - u_{\ell\ell}(s^t)n_t(s^t)] - \theta_t(s^t) = 0, \quad t \geq 1 \\
n_t(s^t): & \quad - (1 + \Phi)u_\ell(s^t) - \Phi [u_{c\ell}(s^t)c_t(s^t) - u_{\ell\ell}(s^t)n_t(s^t)] + \theta_t(s^t) = 0, \quad t \geq 1 
\end{align*}
\] (9.82)

and

\[
\begin{align*}
c_0(s^0, b_0): & \quad (1 + \Phi)u_c(s^0, b_0) + \Phi [u_{cc}(s^0, b_0)c_0(s^0, b_0) - u_{\ell\ell}(s^0, b_0)n_0(s^0, b_0)] - \theta_0(s^0, b_0) \\
& \quad - \Phi u_{cc}(s^0, b_0)b_0 = 0 \\
n_0(s^0, b_0): & \quad - (1 + \Phi)u_\ell(s^0, b_0) - \Phi [u_{c\ell}(s^0, b_0)c_0(s^0, b_0) - u_{\ell\ell}(s^0, b_0)n_0(s^0, b_0)] + \theta_0(s^0, b_0) \\
& \quad + \Phi u_{c\ell}(s^0, b_0)b_0 = 0 
\end{align*}
\] (9.83)

Please note how these first-order conditions differ between \( t = 0 \) and \( t \geq 1 \)

It is instructive to use first-order conditions (9.82) 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

\[
(1 + \Phi)u_c(c, 1 - c - g) + \Phi \left[ cu_{cc}(c, 1 - c - g) - (c + g)u_{\ell\ell}(c, 1 - c - g) \right] = (1 + \Phi)u_\ell(c, 1 - c - g) + \Phi \left[ cu_{c\ell}(c, 1 - c - g) - (c + g)u_{\ell\ell}(c, 1 - c - g) \right] 
\] (9.84)

where we have imposed conditions (9.68) and (9.69)

Equation (9.84) 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
\[(1 + \Phi)u_c(c, 1 - c - g) + \Phi[cu_{cc}(c, 1 - c - g) - (c + g)u_{tc}(c, 1 - c - g)] = (1 + \Phi)u_t(c, 1 - c - g) + \Phi[cu_{ct}(c, 1 - c - g) - (c + g)u_{t\ell}(c, 1 - c - g)] + \Phi(u_{cc} - u_{c,\ell})b_0 \tag{9.85}\]

Notice that a counterpart to \(b_0\) does not appear in (9.84), 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 (9.83) 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 (9.84) 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 \(g\).

**The Ramsey Allocation for a Given \(\Phi\)**

Temporarily take \(\Phi\) as given.

We shall compute \(c_0(s^0, b_0)\) and \(n_0(s^0, b_0)\) from the first-order conditions (9.83).

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 governments 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.

\(\Phi\) has to take a value that assures that the household and the governments budget constraints are both satisfied at a candidate Ramsey allocation and price system associated with that \(\Phi\).

**Further Specialization**

At this point, it is useful to specialize the model in the following ways.

We assume that \(s\) is governed by a finite state Markov chain with states \(s \in [1, \ldots, S]\) and transition matrix \(\Pi\), where

\[\Pi(s'|s) = \text{Prob}(s_{t+1} = s'|s_t = s)\]
Also, assume that government purchases $g$ are an exact time-invariant function $g(s)$ of $s$
We maintain these assumptions throughout the remainder of this lecture

**Determining $\Phi$**

We complete the Ramsey plan by computing the Lagrange multiplier $\Phi$ on the implementability constraint (9.78)

Government budget balance restricts $\Phi$ via the following line of reasoning

The households first-order conditions imply

\[
(1 - \tau_t(s')) = \frac{u_t(s')}{u_c(s')}
\]

(9.86)

and the implied one-period Arrow securities prices

\[
p_{t+1}(s_{t+1}|s^t) = \beta \Pi(s_{t+1}|s_t) \frac{u_c(s^{t+1})}{u_c(s^t)}
\]

(9.87)

Substituting from (9.86), (9.87), and the feasibility condition (9.69) into the recursive version (9.72) of the household budget constraint gives

\[
u_c(s^t)[n_t(s^t) - g_t(s^t)] + \beta \sum_{s_{t+1}} \Pi(s_{t+1}|s_t) u_c(s^{t+1}) b_{t+1}(s_{t+1}|s^t)
\]

\[= u_t(s^t) n_t(s^t) + u_c(s^t) b_t(s_t|s^{t-1})
\]

(9.88)

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 (9.88) while $\beta$ times the conditional expectation of $x_{t+1}(s_t^{t+1})$ 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 (9.88) as

\[
u_c(s)[n(s) - g(s)] + \beta \sum_{s'} \Pi(s'|s) x'(s') = u_t(s) n(s) + x(s)
\]

(9.89)

where $s'$ denotes a next period value of $s$ and $x'(s')$ denotes a next period value of $x$

Equation (9.89) is easy to solve for $x(s)$ for $s = 1, \ldots, S$
If we let $\vec{n}, \vec{g}, \vec{x}$ denote $S \times 1$ vectors whose $i$th elements are the respective $n, g,$ and $x$ values when $s = i$, and let $\Pi$ be the transition matrix for the Markov state $s$, then we can express (9.89) as the matrix equation

$$\vec{u}_c(\vec{n} - \vec{g}) + \beta \Pi \vec{x} = \vec{u}_l \vec{n} + \vec{x}$$

(9.90)

This is a system of $S$ linear equations in the $S \times 1$ vector $x$, whose solution is

$$\vec{x} = (I - \beta \Pi)^{-1} [\vec{u}_c(\vec{n} - \vec{g}) - \vec{u}_l \vec{n}]$$

(9.91)

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

$$\vec{b} = \frac{\vec{x}}{\vec{u}_c}$$

(9.92)

where division here means element-by-element division of the respective components of the $S \times 1$ vectors $\vec{x}$ and $\vec{u}_c$.

Here is a computational algorithm:

1. Start with a guess for the value for $\Phi$, then use the first-order conditions and the feasibility conditions to compute $c(s_t), n(s_t)$ for $s \in [1, \ldots, S]$ and $c_0(s_0, b_0)$ and $n_0(s_0, b_0)$, given $\Phi$
   - these are $2(S + 1)$ equations in $2(S + 1)$ unknowns

2. Solve the $S$ equations (9.91) for the $S$ elements of $\vec{x}$
   - these depend on $\Phi$

3. Find a $\Phi$ that satisfies

$$u_{c,0} b_0 = u_{c,0} (n_0 - g_0) - u_{t,0} n_0 + \beta \sum_{s=1}^{S} \Pi(s|s_0)x(s)$$

(9.93)

by gradually raising $\Phi$ if the left side of (9.93) exceeds the right side and lowering $\Phi$ if the left side is less than the right side

4. After computing a Ramsey allocation, recover the flat tax rate on labor from (9.75) and the implied one-period Arrow securities prices from (9.76)

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 $\Phi$
**Time Inconsistency**

Let \( \{ \tau_t(s^t) \}_{t=0}^{\infty} \) and \( \{ b_{t+1}(s_{t+1}|s^t) \}_{t=0}^{\infty} \) be a time 0, state \( s_0 \) Ramsey plan. Then \( \{ \tau_j(s^j) \}_{j=t}^{\infty} \) and \( \{ b_{j+1}(s_{j+1}|s^j) \}_{j=t}^{\infty} \) 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 \( s_0 \) 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 \( u_{cl} b_t(s_t|s^{t-1}) \) as given, not \( b_t(s_t|s^{t-1}) \). We shall discuss this more below.

**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) [AMSS02], 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 [LS83] 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

\[
u_\ell(c, \ell) \sim -u_n(c, n) \quad u_c(c, \ell) \sim u_n(c, n) \quad u_{\ell,\ell}(c, \ell) \sim u_{nn}(c, n) \quad u_{c,\ell}(c, \ell) \sim u_{nc}(c, n) \quad u_{c,\ell}(c, \ell) \sim 0\]

With these understandings, equations (9.84) and (9.85) simplify in the case of the CRRA utility function. They become
\[(1 + \Phi)[u_c(c) + u_n(c + g)] + \Phi[c u_{cc}(c) + (c + g) u_{nn}(c + g)] = 0 \quad (9.94)\]

and

\[(1 + \Phi)[u_c(c_0) + u_n(c_0 + g_0)] + \Phi[c_0 u_{cc}(c_0) + (c_0 + g_0) u_{nn}(c_0 + g_0)] - \Phi u_{cc}(c_0)b_0 = 0 \quad (9.95)\]

In equation (9.94), 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 \(b_0\):

\[b_0 + g_0 = \tau_0(c_0 + g_0) + \frac{\tilde{b}}{R_0} \quad (9.96)\]

where \(R_0\) is the gross interest rate for the Markov state \(s_0\) that is assumed to prevail at time \(t = 0\) and \(\tau_0\) is the time \(t = 0\) tax rate.

In equation (9.96), it is understood that

\[\tau_0 = 1 - \frac{u_{I,0}}{u_{c,0}}\]

\[R_0 = \beta \sum_{s=1}^{S} \Pi(s|s_0) \frac{u_c(s)}{u_{c,0}}\]

**Sequence Implementation**

The above steps are implemented in a class called `SequentialAllocation`

```python
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 \(\mu\).
    '''

def __init__(self, model):
    # Initialize from model object attributes
    self.\_beta, self.\_pi, self.\_G = model.\_beta, model.\_pi, model.\_G
    self.mc, self.\_Theta = MarkovChain(self.\_pi), model.\_Theta
```

9.4. Optimal Taxation with State-Contingent Debt
self.S = len(model.pi)  # 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:]
    return np.hstack([Uc(c, n) - µ * (Ucc(c, n) * c + Uc(c, n)) - Ξ, Θ * Ξ, 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.')
```python
z = res.x
x = np.linalg.solve(np.eye(S) - self * self, I)
return c, n, x, \Xi

def time0_allocation(self, B_, s_0):
    
    Finds the optimal allocation given initial government debt B_ and state s_0

def FOC(z):
    
    # First order conditions of planner's problem
    return np.hstack([Uc(c, n) * (c - B_) + Un(c, n) * n + \beta * \pi[s_0],
                      Uc(c, n) - \mu * (Ucc(c, n) * (c - B_ + Uc(c, n)) - \Xi,
                      Un(c, n) - \mu * (Unn(c, n) * n + Un(c, n)) + \Theta[s_0] * \Xi,
                      (\Theta * n - c - G)[s_0]])

    # Find root
    res = root(FOC, np.array([0, self.cFB[s_0], self.nFB[s_0], self.EFB[s_0]]))
    if not res.success:
        raise Exception('Could not find time 0 LS allocation."

    return res.x

def timel_value(self, \mu):
    
    # Find the value associated with multiplier \mu
    return c, n, x, V

def T(self, c, n):
    
    # Computes T given c, n
    return model.Uc(c, n), model.Un(c, n)
```

9.4. Optimal Taxation with State-Contingent Debt
9.4.3 Recursive Formulation of the Ramsey problem

\[ x_t(s^t) = u_c(s^t)b_t(s_t|s^{t-1}) \] in equation (9.88) appears to be a purely forward-looking variable

But \( x_t(s^t) \) is also a natural candidate for a state variable in a recursive formulation of the Ramsey problem

**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 has a different objective function and faces different constraints than a Ramsey planner

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
Continuation Ramsey planners do this by choosing continuation policies that induce the representative household to make choices that imply that

\[ u_c(s')b_t(s_t|s_{t-1}) = x_t(s^t) \]

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

A time \( t \geq 1 \) continuation Ramsey planner faces \( x_t, s_t \) as state variables

But the time 0 Ramsey planner faces \( b_0, x_0 \) as state variables

Furthermore, the Ramsey planner cares about \( (c_0(s_0), \ell_0(s_0)) \), while continuation Ramsey planners do not

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

**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 backwards by presenting a Bellman equation for \( V(x, s) \) first, then a Bellman equation for \( W(b, s) \)

**The Continuation Ramsey Problem**

The Bellman equation for a time \( t \geq 1 \) continuation Ramsey planner is

\[
V(x, s) = \max_{n, \{x'(s')\}} u(n - g(s), 1 - n) + \beta \sum_{s' \in S} \Pi(s'|s) V(x', s')
\]

where maximization over \( n \) and the \( S \) elements of \( x'(s') \) is subject to the single implementability constraint for \( t \geq 1 \)

\[
x = u_c(n - g(s)) - u_l n + \beta \sum_{s' \in S} \Pi(s'|s)x'(s')
\]

Here \( u_c \) and \( u_l \) are todays values of the marginal utilities

For each given value of \( x, s \), the continuation Ramsey planner chooses \( n \) and an \( x'(s') \) for each \( s' \in S \)

Associated with a value function \( V(x, s) \) that solves Bellman equation (9.97) are \( S + 1 \) time-invariant policy functions

9.4. Optimal Taxation with State-Contingent Debt
\[ n_t = f(x_t, s_t), \quad t \geq 1 \]
\[ x_{t+1}(s_{t+1}) = h(s_{t+1}; x_t, s_t), \quad s_{t+1} \in S, \quad t \geq 1 \]  
(9.99)

**The Ramsey Problem**

The Bellman equation for the time 0 Ramsey planner is

\[ W(b_0, s_0) = \max_{n_0(x'(s_1))} u(n_0 - g_0, 1 - n_0) + \beta \sum_{s_1 \in S} \Pi(s_1|s_0) V(x'(s_1), s_1) \]  
(9.100)

where maximization over \( n_0 \) and the \( S \) elements of \( x'(s_1) \) is subject to the time 0 implementability constraint

\[ u_c b_0 = u_c(0)(n_0 - g_0) - u_l n_0 + \beta \sum_{s_1 \in S} \Pi(s_1|s_0) x'(s_1) \]  
(9.101)

coming from restriction (9.93)

Associated with a value function \( W(b_0, n_0) \) that solves Bellman equation (9.100) are \( S + 1 \) time 0 policy functions

\[ n_0 = f_0(b_0, s_0) \]
\[ x_1(s_1) = h_0(s_1; b_0, s_0) \]  
(9.102)

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 (9.99) 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 \( E_t \sum_{\tau=1}^{\infty} \beta^{\tau-1} u(c_t, l_t) \), where the consumption and leisure processes are evaluated along the original time 0 Ramsey plan

**First-Order Conditions**

Attach a Lagrange multiplier \( \Phi_1(x, s) \) to constraint (9.98) and a Lagrange multiplier \( \Phi_0 \) to constraint (9.93)

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 planners Bellman equation (9.97) are

\[ \beta \Pi(s'|s)V_x(x', s') - \beta \Pi(s'|s)\Phi_1 = 0 \]  
(9.103)

for \( x'(s') \) and

\[ (1 + \Phi_1)(u_c - u_l) + \Phi_1 [n(u_l - u_c) + (n - g(s))(u_{cc} - u_{lc})] = 0 \]  
(9.104)
for \( n \)

Given \( \Phi_1 \), equation (9.104) is one equation to be solved for \( n \) as a function of \( s \) (or of \( g(s) \))

Equation (9.103) implies \( V_x(x', s') = \Phi_1 \), while an envelope condition is \( V_x(x, s) = \Phi_1 \), so it follows that

\[
V_x(x', s') = V_x(x, s) = \Phi_1(x, s)
\] (9.105)

Time \( t = 0 \): For the time 0 problem on the right side of the Ramsey planners Bellman equation (9.100), first-order conditions are

\[
V_x(x(s_1), s_1) = \Phi_0
\] (9.106)

for \( x(s_1), s_1 \in \mathcal{S} \), and

\[
(1 + \Phi_0)(u_{c,0} - u_{n,0}) + \Phi_0[n_0(u_{ll,0} - u_{lc,0}) + (n_0 - g(s_0))(u_{cc,0} - u_{cl,0}) - \Phi_0(u_{cc,0} - u_{cl,0})b_0 = 0
\] (9.107)

Notice similarities and differences between the first-order conditions for \( t \geq 1 \) and for \( t = 0 \)

An additional term is present in (9.107) 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

**State Variable Degeneracy**

Equations (9.106) and (9.107) imply that \( \Phi_0 = \Phi_1 \) and that

\[
V_x(x_t, s_t) = \Phi_0
\] (9.108)

for all \( t \geq 1 \)

When \( V \) is concave in \( x \), this implies state-variable degeneracy along a Ramsey plan in the sense that for \( t \geq 1 \), \( x_t \) will be a time-invariant function of \( s_t \)

Given \( \Phi_0 \), this function mapping \( s_t \) into \( x_t \) can be expressed as a vector \( \vec{x} \) that solves equation (9.101) for \( n \) and \( c \) as functions of \( g \) that are associated with \( \Phi = \Phi_0 \)

9.4. Optimal Taxation with State-Contingent Debt 1175
Manifestations of Time Inconsistency

While the marginal utility adjusted level of government debt $x_t$ is a key state variable for the continuation Ramsey planners at $t \geq 1$, it is not a state variable at time 0.

The time 0 Ramsey planner faces $b_0$, not $x_0 = u_{c,0}b_0$, as a state variable. The discrepancy in state variables faced by the time 0 Ramsey planner and the time $t \geq 1$ continuation Ramsey planners captures the differing obligations and incentives faced by the time 0 Ramsey planner and the time $t \geq 1$ continuation Ramsey planners:

- The time 0 Ramsey planner is obligated to honor government debt $b_0$ measured in time 0 consumption goods.
- The time 0 Ramsey planner can manipulate the value of government debt as measured by $u_{c,0}b_0$.
- In contrast, time $t \geq 1$ continuation Ramsey planners are obligated not to alter values of debt, as measured by $u_{c,t}b_t$, that they inherit from a preceding Ramsey planner or continuation Ramsey planner.

When government expenditures $g_t$ are a time invariant function of a Markov state $s_t$, a Ramsey plan and associated Ramsey allocation feature marginal utilities of consumption $u_c(s_t)$ that, given $\Phi$, for $t \geq 1$ depend only on $s_t$, but that for $t = 0$ depend on $b_0$ as well.

This means that $u_c(s_t)$ will be a time invariant function of $s_t$ for $t \geq 1$, but except when $b_0 = 0$, a different function for $t = 0$.

This in turn means that prices of one period Arrow securities $p_{t+1}(s_{t+1}|s_t) = p(s_{t+1}|s_t)$ will be the same time invariant functions of $(s_{t+1}, s_t)$ for $t \geq 1$, but a different function $p_0(s_1|s_0)$ for $t = 0$, except when $b_0 = 0$.

The differences between these time 0 and time $t \geq 1$ objects reflect the Ramsey planners incentive to manipulate Arrow security prices and, through them, the value of initial government debt $b_0$.

Recursive Implementation

The above steps are implemented in a class called `RecursiveAllocation`.

```python
from scipy.interpolate import UnivariateSpline
from scipy.optimize import fmin_slsqp

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
```

Chapter 9. Dynamic Programming Squared
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
c, n, x, V = c[ind], n[ind], x[ind], V[ind]  # Sort arrays according to x
        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(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 spprime in range(self.S):
        xprime[spprime] = xprimef[s, spprime](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 spprime in range(self.S):
                    xprime0[spprime] = xprimef[s, spprime](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

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) * θ
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,
    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) + beta * pi[s] @ Vprime)

def cons(z):
c, n, xprime = z[0], z[1], z[2:]
return np.hstack([-Uc(c, n) * c - Un(c, n) * n - beta * pi[s] @ xprime,
                  (Theta * 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, beta, Theta = self.model, self.beta, self.Theta,
    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) + beta * pi[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 - beta * pi[s0] @ xprime,
                  (Theta * n - c - G)[s]])

out, fx, _, imode, smode = fmin_slsqp(objf, self.zFB[s0], f_
if imode > 0:
   raise Exception(smode)

return np.hstack([-fx, out])

9.4.4 Examples

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:
$(0, g_l), (1, g_l), (2, g_l), (3, g_l), (3, g_h), (t \geq 4, g_t)$.

We think of these 6 states as corresponding to $s = 1, 2, 3, 4, 5, 6$.

The transition matrix is

$$
\Pi = \begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.5 \\
0 & 0 & 0 & 0 & 0 & 0.5 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 \\
\end{pmatrix}
$$

Government expenditures at each state are

$$
g = \begin{pmatrix}
0.1 \\
0.1 \\
0.1 \\
0.2 \\
0.1 \\
\end{pmatrix}.
$$

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`

```python
import numpy as np

class CRRAutility:
    def __init__(self, 
        $\beta$=0.9, 
        $\sigma$=2, 
        $\gamma$=2, 
        $\pi$=0.5*np.ones((2, 2)), 
        G=np.array([0.1, 0.2]), 
        $\Theta$=np.ones(2), 
        transfers=False):
        self.$\beta$, self.$\sigma$, self.$\gamma$ = $\beta$, $\sigma$, $\gamma$
        self.$\pi$, self.G, self.$\Theta$, self.transfers = $\pi$, G, $\Theta$, transfers

    # Utility function
    def U(self, c, n):
        $\sigma$ = self.$\sigma$
        if $\sigma$ == 1.:
            U = np.log(c)
        else:
            U = (c**($1 - \sigma$) - 1) / ($1 - \sigma$)
        return U - n**($1 + self.$\gamma$) / ($1 + self.$\gamma$)

    # Derivatives of utility function
    def Uc(self, c, n):
        return c**(-self.$\sigma$)

    def Ucc(self, c, n):
        return -self.$\sigma$ * c**(-self.$\sigma$ - 1)

    def Un(self, c, n):
        return -n**self.$\gamma$

    def Unn(self, c, n):
        return -self.$\gamma$ * n**($self.$\gamma$ - 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
```python
import matplotlib.pyplot as plt

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])

time_θ = np.ones(6)  # Θ can in principle be random

time_example = CRRAutility(π=time_π, G=time_G, θ=time_θ)

time_allocation = SequentialAllocation(time_example)  # Solve sequential problem

tsHist_h = np.array([0, 1, 2, 3, 5, 5])
tsHist_l = np.array([0, 1, 2, 4, 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_h[4] = time_example.G[sHist_h]

# Output paths
sim_seq_h[5] = time_example.θ[sHist_h] * sim_seq_h[1]

fig, axes = plt.subplots(3, 2, figsize=(14, 10))

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()
```

1184 Chapter 9. Dynamic Programming Squared
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 raising the time $t = 0$ value of the gross interest rate for risk-free loans between periods $t$ and $t + 1$, which equals

$$R_t = \frac{u_{c,t}}{\beta E_t[\alpha_{c,t+1}]}$$

A tax policy that makes time $t = 0$ consumption be higher than time $t = 1$ consumption evidently increases the risk-free rate one-period interest rate, $R_t$, at $t = 0$
Raising 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

```python
plt.figure(figsize=(8, 5))
plt.title('Gross Interest Rate')
plt.plot(sim_seq_l[-1], '-ok', sim_seq_h[-1], '-or', alpha=0.7)
plt.grid()
plt.show()
```

**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 $g_3 = g_t$
• 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 level required to service the interest payments on the debt and government expenditures.

**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 raising a risk-free interest rate for one-period loans between times $t = 0$ and $t = 1$.

By raising 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.

**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 $g_t = g_t$ 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).

```python
tax_sequence = SequentialAllocation(CRRAutility(G=0.15, 
                                       \pi=np.ones((1, 1)), 
                                       \Theta=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()
```

9.4. Optimal Taxation with State-Contingent Debt
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 raises the one-period interest rate at time 0 above that of subsequent periods. There are only two values of initial government debt at which the tax rate is constant for all $t \geq 0$. 

![Graph showing tax rate and gross interest rate](image)
The first is $b_0 = 0$

- Here the government can't use the $t = 0$ tax rate to alter the value of the initial debt.

The second occurs when the government enters with sufficiently large assets that the Ramsey planner can achieve first best and sets $\tau_t = 0$ for all $t$.

It is only for these two values of initial government debt that the Ramsey plan is time-consistent.

Another way of saying this is that, except for these two values of initial government debt, a continuation of a Ramsey plan is not a Ramsey plan.

To illustrate this, consider a Ramsey planner who starts with an initial government debt $b_1$ associated with one of the Ramsey plans computed above.

Call $\tau_1^R$ the time $t = 0$ tax rate chosen by the Ramsey planner confronting this value for initial government debt.

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.

```python
# Pseudocode for illustrating the Ramsey plan

tax_sequence = SequentialAllocation(CRRAutility(G=0.15,
                                               \[\pi=\text{np.ones}((1, 1)),
                                                  \Theta=\text{np.ones}(1)\])

n = 100
tax_policy = np.empty((n, 2))
t_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]
    t_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, t_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

**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 $U_{cc} = -\sigma U_c$ and $U_{nn} = \gamma U_n$ 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.
class LogUtility:
    def __init__(self, 
        beta=0.9, 
        psi=0.69, 
        pi=0.5*np.ones((2, 2)), 
        G=np.array([0.1, 0.2]), 
        Theta=np.ones(2), 
        transfers=False):
        self.beta, self.psi, self.pi = beta, psi, pi
        self.G, self Theta, self.transfers = G, Theta, transfers

    # Utility function
    def U(self, c, n):
        return np.log(c) + self.psi + 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.psi / (1 - n)

    def Unn(self, c, n):
        return -self.psi / (1 - n)**2

Also suppose that $g_t$ follows a two state i.i.d. 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

```python
log_example = LogUtility()
seq_log = SequentialAllocation(log_example)  # Solve sequential problem

# Initialize grid for value function iteration and solve
mu_grid = np.linspace(-0.6, 0.0, 200)
bel_log = RecursiveAllocation(log_example, mu_grid)  # Solve recursive problem

T = 20
shist = np.array([0, 0, 0, 0, 0, 0, 
                  0, 1, 1, 0, 0, 0, 1, 
                  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
```

9.4. Optimal Taxation with State-Contingent Debt
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.
Further Comments

A related lecture describes an extension of the Lucas-Stokey model by Aiyagari, Marcet, Sargent, and Seppälä (2002) [AMSS02]

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

9.5 Optimal Taxation without State-Contingent Debt

Contents

- Optimal Taxation without State-Contingent Debt
  - Overview
  - Competitive Equilibrium with Distorting Taxes
  - Recursive Version of AMSS Model
  - Examples

9.5.1 Overview

In an earlier lecture we described a model of optimal taxation with state-contingent debt due to Robert E. Lucas, Jr., and Nancy Stokey [LS83]

Aiyagari, Marcet, Sargent, and Seppälä [AMSS02] (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
9.5.2 Competitive Equilibrium with Distorting Taxes

Many but not all features of the economy are identical to those of the Lucas-Stokey economy. Let's start with things that are identical.

For $t \geq 0$, a history of the state is represented by $s^t = [s_t, s_{t-1}, \ldots, s_0]$. Government purchases $g(s)$ are an exact time-invariant function of $s$.

Let $c_t(s^t)$, $\ell_t(s^t)$, and $n_t(s^t)$ denote consumption, leisure, and labor supply, respectively, at history $s^t$ at time $t$.

Each period a representative household is endowed with one unit of time that can be divided between leisure $\ell_t$ and labor $n_t$:

$$n_t(s^t) + \ell_t(s^t) = 1 \quad (9.109)$$

Output equals $n_t(s^t)$ and can be divided between consumption $c_t(s^t)$ and $g(s_t)$:

$$c_t(s^t) + g(s_t) = n_t(s^t) \quad (9.110)$$

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

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)] \quad (9.111)$$

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 [AMSS02] 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 [Bar79] than they do in [LS83].
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 nonnegative lump-sum transfer to the representative household

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 $b_t(s^t)$ divided by $R_t(s^t)$.

The governments budget constraint in period $t$ at history $s^t$ is

$$b_t(s^{t-1}) = \tau_t(s^t) n_t(s^t) - g_t(s^t) - T_t(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)} \equiv z(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)},$$

(9.112)

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 governments budget constraint (9.112) yields:

$$b_t(s^{t-1}) = z(s^t) + \beta \sum_{s^{t+1}|s^t} \pi_{t+1}(s^{t+1}|s^t) \frac{u_c(s^{t+1})}{u_c(s^t)} - b_{t+1}(s^t).$$

(9.113)

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 (9.113) 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.

---

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.
If we replace $b_{t+1}(s^t)$ on the right side of equation (9.113) by the right side of next periods budget constraint (associated with a particular realization $s_t$) we get

$$b_t(s^{t-1}) = z(s^t) + \sum_{s^{t+1}|s^t} \beta \pi_{t+1}(s^{t+1}|s^t) \frac{u_e(s^{t+1})}{u_c(s^t)} \left[ z(s^{t+1}) + \frac{b_{t+2}(s^{t+1})}{R_{t+1}(s^{t+1})} \right]$$

After making similar repeated substitutions for all future occurrences of government indebtedness, and by invoking the natural debt limit, we arrive at:

$$b_t(s^{t-1}) = \sum_{j=0}^{\infty} \sum_{s^{t+j}|s^t} \beta^j \pi_{t+j}(s^{t+j}|s^t) \frac{u_e(s^{t+j})}{u_c(s^t)} z(s^{t+j}) \quad (9.114)$$

Now let's

- substitute the resource constraint into the net-of-interest government surplus, and
- use the households first-order condition $1 - \pi_t^n(s^t) = u_e(s^t)/u_c(s^t)$ to eliminate the labor tax rate so that we can express the net-of-interest government surplus $z(s^t)$ as

$$z(s^t) = \left[ 1 - \frac{u_e(s^t)}{u_c(s^t)} \right] \left[ c_t(s^t) + g_t(s_t) \right] - g_t(s_t) - T_t(s^t). \quad (9.115)$$

If we substitute the appropriate versions of right side of (9.115) for $z(s^{t+j})$ into equation (9.114), we obtain a sequence of implementability constraints on a Ramsey allocation in an AMSS economy.

Expression (9.114) at time $t = 0$ and initial state $s^0$ was also an implementability constraint on a Ramsey allocation in a Lucas-Stokey economy:

$$b_0(s^{-1}) = \mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_e(s^j)}{u_c(s^0)} z(s^j) \quad (9.116)$$

Indeed, it was the only implementability constraint there.

But now we also have a large number of additional implementability constraints

$$b_t(s^{t-1}) = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_e(s^{t+j})}{u_c(s^t)} z(s^{t+j}) \quad (9.117)$$

Equation (9.117) must hold for each $s^t$ for each $t \geq 1$.

**Comparison with Lucas-Stokey Economy**

The expression on the right side of (9.117) 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 $t$. 

1196 Chapter 9. Dynamic Programming Squared
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.

**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

$$
\max_{\{c_t(s^t), b_{t+1}(s^t)\}} \mathbb{E}_t \sum_{t=0}^{\infty} \beta^t u\left(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)\right)
$$

where the maximization is subject to

$$
\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} z(s^{t+j}) \geq b_0(s^{-1})
$$

(9.118)

and

$$
\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} z(s^{t+j}) = b_t(s^{t-1}) \quad \forall s^t
$$

(9.119)

given $b_0(s^{-1})$.

**Lagrangian Formulation**

Let $\gamma_0(s^0)$ be a nonnegative Lagrange multiplier on constraint (9.118)

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 (9.119).

Depending on how the constraints bind, these multipliers can be positive or negative:

$$
\gamma_t(s^t) \geq (\leq) 0 \quad \text{if the constraint binds in this direction}
$$

$$
\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} z(s^{t+j}) \geq (\leq) b_t(s^{t-1}).
$$

9.5. Optimal Taxation without State-Contingent Debt
A negative multiplier $\gamma_t(s^t) < 0$ means that if we could relax constraint (9.119), we would like to increase the beginning-of-period indebtedness for that particular realization of history $s^t$

That would let us reduce the beginning-of-period indebtedness for some other history.$^2$

These features flow from the fact that the government cannot use state-contingent debt and therefore cannot allocate its indebtedness efficiently across future states.

**Some Calculations**

It is helpful to apply two transformations to the Lagrangian

Multiply constraint (9.118) by $u_c(s^0)$ and the constraints (9.119) by $\beta^t u_c(s^t)$

Then a Lagrangian for the Ramsey problem can be represented as

\[
J = \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left\{ u(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)) + \gamma_t(s^t) \left[ \mathbb{E} \sum_{j=0}^{\infty} \beta^j u_c(s^{t+j}) z(s^{t+j}) - u_c(s^t) b_t(s^{t-1}) \right] \right\} 
\]

(9.120)

where

\[
\Psi_t(s^t) = \Psi_{t-1}(s^{t-1}) + \gamma_t(s^t) \quad \text{and} \quad \Psi_{-1}(s^{-1}) = 0
\]

(9.121)

In (9.120), the second equality uses the law of iterated expectations and Abels summation formula (also called summation by parts, see this page)

First-order conditions with respect to $c_t(s^t)$ can be expressed as

\[
u_c(s^t) - u_c(s^t) + \Psi_t(s^t) \left\{ \left[ u_{cc}(s^t) - u_{c}(s^t) \right] z(s^t) + u_c(s^t) z_c(s^t) \right\} - \gamma_t(s^t) \left[ u_{cc}(s^t) - u_{c}(s^t) \right] b_t(s^{t-1}) = 0
\]

(9.122)

and with respect to $b_t(s^t)$ as

\[
\mathbb{E}_t \left[ \gamma_{t+1}(s^{t+1}) u_c(s^{t+1}) \right] = 0
\]

(9.123)

$^2$ 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 (9.119) takes a positive value, so $\gamma_t(\tilde{s}^t) > 0$. 

---

Chapter 9. Dynamic Programming Squared
If we substitute $z(s^t)$ from (9.115) and its derivative $z_c(s^t)$ into first-order condition (9.122), 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 first-order condition (9.122) 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 periods state, a constraint that would not be present if government debt could be state contingent

2. The Lagrange multiplier $\Psi_t(s^t)$ in first-order condition (9.122) may change over time in response to realizations of the state, while the multiplier $\Phi$ in the Lucas-Stokey economy is time invariant

We need some code from our *an earlier lecture* on optimal taxation with state-contingent debt sequential allocation implementation:

```python
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 $\mu$.
    '''

    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])
```

9.5. Optimal Taxation without State-Contingent Debt
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.zFB = np.hstack([self.cFB, self.nFB, self.zFB])

def time1_allocation(self):
    '''
    Computes optimal allocation for time t >= 1 for a given 
    '''
    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:]
    return np.hstack([Uc(c, n) - μ * (Ucc(c, n) * c + Uc(c, n)) - Ξ, Θ * Ξ])
    # FOC of c
    Θ * n - c - G)
    # FOC of n

# 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

z = np.hstack([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):
        c = z[:S]
        n = z[S:2 * S]
        Ξ = z[2 * S:]
        return np.hstack([Uc(c, n) - μ * (Ucc(c, n) * c + Uc(c, n)) - Ξ, Θ * Ξ])
        # FOC of c
        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, x, Ξ = np.hstack([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

# Multiplier on the resource constraint
self.zFB = np.hstack([self.cFB, self.nFB, self.zFB])
\[ \mu, c, n, \Xi = z \]
\[ xprime = \text{self.time1_allocation}(\mu)[2] \]
\[ \text{return np.hstack([Uc(c, n) * (c - B_) + Un(c, n) * n + } \beta * \pi[s_0], \]
\[ \approx \text{xprime}, \]
\[ Uc(c, n) - \mu * (Ucc(c, n) * (c - B_) + Uc(c, n)) - \Xi, \]
\[ Un(c, n) - \mu * (Unn(c, n) * n + Un(c, n)) + \Theta[s_0] + \Xi, \]
\[ (\Theta * n - c - G)[s_0]) \]

# Find root
res = root(FOC, np.array([0, self.cFB[s_0], self.nFB[s_0], self.\XiFB[s_0]]))
if not res.success:
    raise Exception('Could not find time 0 LS allocation.')
return res.x

def time1_value(self, mu):
    '''
    Find the value associated with multiplier \mu
    '''
    c, n, x, \Xi = self.time1_allocation(mu)
    U = self.model.U(c, n)
    V = np.linalg.solve(np.eye(self.S) - self.\beta * self.\pi, 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.\Theta * Uc)

def simulate(self, B_, s_0, T, sHist=None):
    '''
    Simulates planners policies for T periods
    '''
    model, \pi, \beta = self.model, self.\pi, self.\beta
    Uc = model.Uc
    if sHist is None:
        sHist = self.mc.simulate(T, s_0)
    cHist, nHist, Bhist, THist, \muHist = np.zeros((5, T))
    RHist = np.zeros(T - 1)

    # Time 0
    \mu, cHist[0], nHist[0], _ = self.time0_allocation(B_, s_0)
    THist[0] = self.T(cHist[0], nHist[0])[s_0]
    Bhist[0] = B_
To analyze the AMSS model, we find it useful to adopt a recursive formulation using techniques like those in our lectures on \textit{dynamic Stackelberg models} and \textit{optimal taxation with state-contingent debt}.

### 9.5.3 Recursive Version of AMSS Model

We now describe a recursive formulation of the AMSS economy.

We have noted that from the point of view of the Ramsey planner, the restriction to one-period risk-free securities

- leaves intact the single implementability constraint on allocations (9.116) from the Lucas-Stokey economy, but
- adds measurability constraints (9.114) 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.

#### Recasting State Variables

In the AMSS setting, the government faces a sequence of budget constraints

\[
\pi_t(s^t)n_t(s^t) + T_t(s^t) + b_{t+1}(s^t)/R_t(s^t) = g_t + b_t(s^{t-1})
\]

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 nonnegative 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 [AMSS02], who restricted transfers to be nonnegative).

In this case, the household faces a sequence of budget constraints

\[
b_t(s^{t-1}) + (1 - \pi_t(s^t))n_t(s^t) = c_t(s^t) + b_{t+1}(s^t)/R_t(s^t)
\]
The households first-order conditions are $u_{c,t} = \beta R_t E_t u_{c,t+1}$ and $(1 - \tau_t)u_{c,t} = u_{t,t}$

Using these to eliminate $R_t$ and $\tau_t$ from budget constraint (9.124) gives

$$b_t(s^{t-1}) + \frac{u_{t,t}(s^t_n)}{u_{c,t}(s^t_n)} n_t(s^t_n) = c_t(s^t_n) + \frac{\beta (E_t u_{c,t+1}) b_{t+1}(s^t_n)}{u_{c,t}(s^t_n)}$$

(9.125)

or

$$u_{c,t}(s^t_n) b_t(s^{t-1}) + u_{t,t}(s^t_n) n_t(s^t_n) = c_t(s^t_n) + \beta (E_t u_{c,t+1}) b_{t+1}(s^t_n)$$

(9.126)

Now define

$$x_t \equiv \beta b_{t+1}(s^t) E_t u_{c,t+1} = u_{c,t}(s^t) \frac{b_{t+1}(s^t)}{R_t(s^t)}$$

(9.127)

and represent the households budget constraint at time $t$, history $s^t$ as

$$\frac{u_{c,t} x_{t-1}}{\beta E_{t-1} u_{c,t}} = u_{c,t} c_t - u_{t,t} n_t + x_t$$

(9.128)

for $t \geq 1$

**Measurability Constraints**

Write equation (9.126) as

$$b_t(s^{t-1}) = c_t(s^t_n) - \frac{u_{t,t}(s^t_n)}{u_{c,t}(s^t_n)} n_t(s^t_n) + \frac{\beta (E_t u_{c,t+1}) b_{t+1}(s^t_n)}{u_{c,t}}$$

(9.129)

The right side of equation (9.129) 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 (9.129) must equal $b_t(s^{t-1})$

That implies that it is has to be measurable with respect to $s^{t-1}$

Equations (9.129) are the *measurability constraints* that the AMSS model adds to the single time 0 implementation constraint imposed in the Lucas and Stokey model

**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

$$V(x, s) = \max_{\{n(s), x(s)\}} \sum_s \Pi(s|s_\cdot) [u(n(s) - g(s), 1 - n(s)) + \beta V(x(s), s)]$$  

subject to the following collection of implementability constraints, one for each $s \in S$:

$$\frac{u_c(s)x}{\beta \sum_s \Pi(s|s_\cdot)u_c(s)} = u_c(s)(n(s) - g(s)) - u_t(s)n(s) + x(s)$$  

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

$$W(b_0, s_0) = \max_{n_0, x_0} u(n_0 - g_0, 1 - n_0) + \beta V(x_0, s_0)$$  

where maximization is subject to

$$u_{c,0}b_0 = u_{c,0}(n_0 - g_0) - u_{t,0}n_0 + x_0$$  

**Martingale Supercedes State-Variable Degeneracy**

Let $\mu(s|s_\cdot)\Pi(s|s_\cdot)$ be a Lagrange multiplier on constraint (9.131) for state $s$.

After forming an appropriate Lagrangian, we find that the continuation Ramsey planners first-order condition with respect to $x(s)$ is

$$\beta V_x(x(s), s) = \mu(s|s_\cdot)$$  

Applying the envelope theorem to Bellman equation (9.130) gives

$$V_x(x, s) = \sum_s \Pi(s|s_\cdot)\mu(s|s_\cdot) \frac{u_c(s)}{\beta \sum_s \Pi(s|s_\cdot)u_c(s)}$$
Equations (9.134) and (9.135) imply that

\[ V_x(x_-, s_-) = \sum_s \left( \Pi(s|s_-) \frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}|s_-)u_c(\tilde{s})} \right) V_x(x(s), s) \]  

(9.136)

Equation (9.136) 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 is generated by the twisted transition probability matrix:

\[ \bar{\Pi}(s|s_-) \equiv \Pi(s|s_-) \frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}|s_-)u_c(\tilde{s})} \]

Exercise: Please verify that \( \bar{\Pi}(s|s_-) \) is a valid Markov transition density, i.e., that its elements are all nonnegative and that for each \( s_- \), the sum over \( s \) equals unity

**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

**Digression on Nonnegative Transfers**

Throughout this lecture we have imposed that transfers \( T_t = 0 \)

AMSS [*AMSS02*] 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

\[
(1 - \mu(s|s_-))(1 - u_l(s)) + \mu(s|s_-)n(s)u_H(s) = 0
\]

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 nonnegative lump sum transfers

**Code**

The recursive formulation is implemented as follows

```python
from scipy.optimize import fmin_slsqp

class RecursiveAllocationAMSS:

    def __init__(self, model, grid, tol_diff=1e-4, tol=1e-4):
        self.\beta, self.\pi, self.G = model.\beta, model.\pi, model.G
        self.mc, self.S = MarkovChain(self.\pi), len(model.\pi)  # Number of states

    def solve_time1_bellman(self):
        '''
        Solve the time 1 Bellman equation for calibration model and initial grid \mugrid0
        '''
        model, \mugrid0 = self.model, self.\mugrid
        \pi = model.\pi
        S = len(model.\pi)

        # 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(\mu_, s_):
            c, n, x, V = PP.time1_value(\mu_)
            return c, n, \pi[s_] @ x, \pi[s_] @ V

        cf, nf, xgrid, Vf, xprimef = [], [], [], [], []
        for s_ in range(S):
            c, n, x, V = zip(*map(lambda \mu: incomplete_allocation(\mu, s_), \mugrid0))
            c, n = np.vstack(c).T, np.vstack(n).T
```

1206 Chapter 9. Dynamic Programming Squared
x, V = np.hstack(x), np.hstack(V)
xprimes = np.vstack([x] * S)
cf.append(interp(x, c))
np.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.μ), 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

9.5. Optimal Taxation without State-Contingent Debt 1207
```python
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, μHist = np.zeros((7, T))
    # time 0
    cHist[0], nHist[0], xHist[0], THist[0] = self.time0_allocation(B_, 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[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, μHist, sHist, xHist])
```

1208 Chapter 9. Dynamic Programming Squared
class BellmanEquation:
    '''
    Bellman equation for the continuation of the Lucas-Stokey Problem
    '''

    def __init__(self, model, xgrid, policies0, tol, maxiter=1000):
        self.\beta, self.\pi, self.G = model.\beta, model.\pi, model.G
        self.S = \text{len}(model.\pi)  # Number of states
        self.\Theta, self.model, self.tol = model.\Theta, 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, \Theta, Uc, Un, G = self.S, self.\Theta, model.Uc, model.Un, self.G

        def res(z):
            c = z[:S]
            n = z[S:]
            return np.hstack([\Theta \ast Uc(c, n) + Un(c, n), \Theta \ast n - c - G])

        res = root(res, 0.5 \ast np.ones(2 \ast 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) \ast self.cFB + \n              Un(self.cFB, self.nFB) \ast self.nFB

        self.xFB = np.linalg.solve(np.eye(S) - self.\beta \ast self.\pi, IFB)
        self.zFB = {}
```python
for s in range(S):
    self.zFB[s] = np.hstack([self.cFB[s], self.nFB[s], self.pi[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
        '''
        u, Uc, Un = self.U, self.Uc, self.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 -pi[s_] @ (U(c, n) + beta * 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 = pi[s_] @ u_c
            return np.hstack([x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - beta * xprime,
                              theta * 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., 100.0)]
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])

9.5.4 Examples

We now turn to some examples

We will first build some useful functions for solving the model

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)
```python
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
```

**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:

```python
import numpy as np

class CRRAutility:
    def __init__(self, 
                 \( \beta \)=0.9, 
                 \( \sigma \)=2, 
                 \( \gamma \)=2, 
                 \( \pi \)=0.5*np.ones((2, 2)), 
                 G=np.array([0.1, 0.2]), 
                 \( \Theta \)=np.ones(2), 
                 transfers=False):
        self.\beta, self.\sigma, self.\gamma = \beta, \sigma, \gamma
        self.\pi, self.G, self.\Theta, self.transfers = \pi, G, \Theta, transfers

    # Utility function
    def U(self, c, n):
        \( \sigma = self.\sigma \)
        if \( \sigma == 1. \):
            U = np.log(c)
        else:
            U = (c**((1 - \sigma) - 1)) / (1 - \sigma)
        return U - n**((1 + self.\gamma) / (1 + self.\gamma))

    # Derivatives of utility function
    def Uc(self, c, n):
        return c**(-self.\sigma)

    def Ucc(self, c, n):
        return -self.\sigma * c**(-self.\sigma - 1)

    def Un(self, c, n):
        return -n**self.\gamma

    def Unn(self, c, n):
        return -self.\gamma * n**(self.\gamma - 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.

```python
import matplotlib.pyplot as plt

# Initialize grid 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, 0.5, 0.5, 0, 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

time_sequential = SequentialAllocation(time_example)  # Solve sequential problem

time_bellman = RecursiveAllocationAMSS(time_example, μ_grid)  # Solve recursive problem

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_h[4] = time_example.G[sHist_h]
sim_bel_h[4] = time_example.G[sHist_h]

# Output paths
sim_seq_h[5] = time_example.Θ[sHist_h] * sim_seq_h[1]

fig, axes = plt.subplots(3, 2, figsize=(14, 10))
```
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 facilitates 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

\[
\begin{align*}
u(c, n) &= \log(c) + 0.69 \log(1 - n) 
\end{align*}
\]

In accordance, we will re-define our utility function

```python
class LogUtility:
    def __init__(self,
                 \beta=0.9,
                 \psi=0.69,
                 \pi=0.5*\text{np.ones}(2, 2),
                 G=\text{np.array}([0.1, 0.2]),
                 \Theta=\text{np.ones}(2),
                 transfers=False):
        self.\beta, self.\psi, self.\pi = \beta, \psi, \pi
        self.G, self.\Theta, self.transfers = G, \Theta, transfers

    # Utility function
    def U(self, c, n):
        return \text{np.log}(c) + self.\psi * \text{np.log}(1 - n)

    # Derivatives of utility function
    def Uc(self, c, n):
        return 1 / c
```

9.5. Optimal Taxation without State-Contingent Debt 1217
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).

```python
log_example = LogUtility()
log_example.transfers = True  # Government can use transfers
log_sequential = SequentialAllocation(log_example)  # Solve sequential problem
log_bellman = RecursiveAllocationAMSS(log_example, _grid)

T = 20
sHist = np.array([0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 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)


# Government spending paths
sim_seq[4] = log_example.G[sHist]

# Output paths

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()
```
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.

```python
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)

# Government spending paths
sim_seq_long[4] = log_example.G[sHist_long]

# Output paths
```
Figures are shown illustrating the impact of fluctuating interest rates on various economic indicators such as consumption, labor supply, government debt, tax rate, government spending, and output. The plots demonstrate how fiscal insurance can be achieved through varying interest rates in dynamic economic models.

### 9.6 Fluctuating Interest Rates Deliver Fiscal Insurance

**Contents**

- Fluctuating Interest Rates Deliver Fiscal Insurance
  - Overview
Co-authors: Anmol Bhandari and David Evans

9.6.1 Overview

This lecture extends our investigations of how optimal policies for levying a flat-rate tax on labor income and issuing government debt depend on whether there are complete markets for debt.

A Ramsey allocation and Ramsey policy in the AMSS [AMSS02] model described in *optimal taxation without state-contingent debt* generally differs from a Ramsey allocation and Ramsey policy in the Lucas-Stokey [LS83] model described in *optimal taxation with state-contingent debt*.

This is because the implementability restriction that a competitive equilibrium with a distorting tax imposes on allocations in the Lucas-Stokey model is just one among a set of implementability conditions imposed in the AMSS model.

These additional constraints require that time $t$ components of a Ramsey allocation for the AMSS model be measurable with respect to time $\tilde{t} - 1$ 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 [LS83] model with state-contingent debt.

Another way to say this is that differences between the Ramsey allocations of the two models indicate that some of the measurability constraints 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 . . .

9.6. Fluctuating Interest Rates Deliver Fiscal Insurance
– 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 (9.116) converge to zero

• After the implementability constraints (9.116) 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 $t \geq T$
  – $\bar{b} < 0$, 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 [LS83] 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 government assets, fluctuations in the one-period risk-free interest rate provide the government with complete insurance against stochastically varying government expenditures

### 9.6.2 Forces at work

The forces driving asymptotic outcomes here are examples of dynamics present in a more general class incomplete markets models analyzed in [BEGS17] (BEGS)
BEGS provide conditions under which government debt under a Ramsey plan converges to an invariant distribution.

BEGS construct approximations to that asymptotically invariant distribution of government debt under a Ramsey plan.

BEGS also compute an approximation to a Ramsey plan’s rate of convergence to that limiting invariant distribution.

We shall use the BEGS approximating limiting distribution and 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.

**9.6.3 Logical flow of lecture**

We present ideas in the following order:

- We describe a two-state AMSS economy and generate a long simulation starting from a positive initial government debt.

- We observe that in a long simulation starting from positive government debt, the par value of government debt eventually converges to a constant $b$.

- 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

**Equations from Lucas-Stokey (1983) model**

Although we are studying an AMSS [AMSS02] economy, a Lucas-Stokey [LS83] economy plays an important role in the reverse-engineering calculation to be described below.

For that reason, it is helpful to have 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

\[
(1 + \Phi)u_c(c, 1 - c - g) + \Phi \left[ cu_{cc}(c, 1 - c - g) - (c + g)u_{\ell c}(c, 1 - c - g) \right] = (1 + \Phi)u_{\ell}(c, 1 - c - g) + \Phi \left[ cu_{\ell \ell}(c, 1 - c - g) - (c + g)u_{\ell \ell}(c, 1 - c - g) \right] + \Phi (u_{cc} - u_{c,\ell})b_0
\]

There is one such equation for each value of the Markov state \( s_0 \).

In addition, given an initial Markov state, the time \( t = 0 \) quantities \( c_0 \) and \( b_0 \) satisfy

\[
(1 + \Phi)u_c(c, 1 - c - g) + \Phi \left[ cu_{cc}(c, 1 - c - g) - (c + g)u_{\ell c}(c, 1 - c - g) \right] = (1 + \Phi)u_{\ell}(c, 1 - c - g) + \Phi \left[ cu_{\ell \ell}(c, 1 - c - g) - (c + g)u_{\ell \ell}(c, 1 - c - g) \right] + \Phi (u_{cc} - u_{c,\ell})b_0
\]

In addition, the time \( t = 0 \) budget constraint is satisfied at \( c_0 \) and initial government debt \( b_0 \):

\[
b_0 + g_0 = \tau_0(c_0 + g_0) + \frac{\bar{b}}{R_0} \tag{9.139}
\]

where \( R_0 \) is the gross interest rate for the Markov state \( s_0 \) that is assumed to prevail at time \( t = 0 \) and \( \tau_0 \) is the time \( t = 0 \) tax rate.

In equation (9.139), it is understood that

\[
\tau_0 = 1 - \frac{u_{t,0}}{u_{c,0}}
\]

\[
R_0^{-1} = \beta \sum_{s=1}^{S} \Pi(s | s_0) \frac{u_{c}(s)}{u_{c,0}}
\]

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.
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

\[ u_\ell(c, \ell) \sim -u_\alpha(c, n) \]
\[ u_c(c, \ell) \sim u_c(c, n) \]
\[ u_{\ell,\ell}(c, \ell) \sim u_{\alpha\alpha}(c, n) \]
\[ u_{c,c}(c, \ell) \sim u_{c,c}(c, n) \]
\[ u_{c,\ell}(c, \ell) \sim 0 \]

With these understandings, equations (9.137) and (9.138) simplify in the case of the CRRA utility function.

They become

\[ (1 + \Phi)[u_c(c) + u_\alpha(c + g)] + \Phi[cu_{c,c}(c) + (c + g)u_{\alpha\alpha}(c + g)] = 0 \quad (9.140) \]

and

\[ (1 + \Phi)[u_c(c_0) + u_\alpha(c_0 + g_0)] + \Phi[c_0u_{c,c}(c_0) + (c_0 + g_0)u_{\alpha\alpha}(c_0 + g_0)] - \Phi u_{c,c}(c_0)b_0 = 0 \quad (9.141) \]

In equation (9.140), 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

```python
import numpy as np

class CRRAutility:
    def __init__(self,
        \( \beta = 0.9 \),
        \( \sigma = 2 \),
        \( \gamma = 2 \),
        \( \pi = 0.5 \times \text{np.ones((2, 2))} \),
        \( G = \text{np.array([0.1, 0.2])} \),
        \( \Theta = \text{np.ones(2)} \),
        transfers=False):
```

9.6. Fluctuating Interest Rates Deliver Fiscal Insurance 1225
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)

9.6.4 Example economy

We set the following parameter values
The Markov state $s_t$ takes two values, namely, 0, 1
The initial Markov state is 0
The Markov transition matrix is $0.5I$ where $I$ is a $2 \times 2$ identity matrix, so the $s_t$ process is i.i.d.
Government expenditures $g(s)$ equal .1 in Markov state 0 and .2 in Markov state 1
We set preference parameters as follows:

$\beta = .9$
$\sigma = 2$
$\gamma = 2$

Here are several classes that do most of the work for us
The code is mostly taken or adapted from the earlier lectures optimal taxation without state-contingent debt
and optimal taxation with state-contingent debt

import numpy as np
from scipy.optimize import root
from quantecon import MarkovChain
class SequentialAllocation:

    '''
    Class that takes CESutility or BGPutility object as input returns
    planner's allocation as a function of the multiplier on the
    implementability constraint μ.
    '''

    def __init__(self, model):
        # Initialize from model object attributes
        self.β, self.π, self.G = model.β, model.π, model.G
        self.mc, self.Θ = MarkovChain(self.π), model.Θ
        self.S = len(model.π)  # Number of states
        self.model = model

        # Find the first best allocation
        self.find_first_best()

def find_first_best(self):
    '''
    Find the first best allocation
    '''
    model = self.model
    S, Θ, G = self.S, self.Θ, self.G
    Uc, Un = model.Uc, model.Un

    def res(z):
        c = z[:S]
        n = z[S:]
        return np.hstack([Θ + Uc(c, n) + Un(c, n), Θ * n - c - G])

    res = root(res, 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 timel 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:]
    return np.hstack([Uc(c, n) - μ * (Ucc(c, n) * c + Uc(c, n)) - Ξ, Θ * Ξ, Θ * n - c - G])

# Find 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
z = z[:S], z[S:2*S], z[2*S:]

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],
                           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):
    
    '1228 Chapter 9. Dynamic Programming Squared'
Find the value associated with multiplier $\mu$

```python
c, n, x, \Xi = self.time1_allocation(\mu)
U = self.model.U(c, n)
V = np.linalg.solve(np.eye(self.S) - self.\beta * self.\pi, U)
return c, n, x, V
```

```python
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.\Theta + Uc)
```

```python
def simulate(self, B_, s_0, T, sHist=None):
    
    Simulates planners policies for $T$ periods
    model, \pi, \beta = self.model, self.\pi, self.\beta
    Uc = model.Uc
    if sHist is None:
        sHist = self.mc.simulate(T, s_0)
    cHist, nHist, Bhist, THist, \muHist = np.zeros((5, T))
    RHist = np.zeros(T - 1)
    # Time 0
    \mu, cHist[0], nHist[0], _ = self.time0_allocation(B_, s_0)
    THist[0] = self.T(cHist[0], nHist[0])[s_0]
    Bhist[0] = B_
    \muHist[0] = \mu
    # Time 1 onward
    for t in range(1, T):
        c, n, x, \Xi = self.time1_allocation(\mu)
        T = self.T(c, n)
        u_c = Uc(c, n)
        s = sHist[t]
        Eu_c = \pi[sHist[t - 1]] @ u_c
        cHist[t], nHist[t], Bhist[t], THist[t] = c[s], n[s], x[s] / \n            u_c[s], T[s]
        RHist[t - 1] = Uc(cHist[t - 1], nHist[t - 1]) / (\beta * Eu_c)
        \muHist[t] = \mu
    return np.array([cHist, nHist, Bhist, THist, sHist, \muHist, RHist])
```

```python
from scipy.optimize import fmin_slsqp

class RecursiveAllocationAMSS:
```

9.6. Fluctuating Interest Rates Deliver Fiscal Insurance
def __init__(self, model, grid, 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_), μ, 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
```python
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.pi), 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
```

9.6. Fluctuating Interest Rates Deliver Fiscal Insurance
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, Hist = np.zeros((7, T))
    # time 0
    cHist[0], nHist[0], xHist[0], THist[0] = self.time0_allocation(B_, 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, Hist, 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

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):

9.6. Fluctuating Interest Rates Deliver Fiscal Insurance 1233
Finds the optimal policies


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 -(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[:S]

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,
        (∑ * 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])

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:
**9.6.5 Reverse engineering strategy**

We can reverse engineer a value $b_0$ of initial debt due that renders the AMSS measurability constraints not binding from time $t = 0$ onward.

We accomplish this by recognizing that if the AMSS measurability constraints never bind, then the AMSS allocation and Ramsey plan is equivalent with that for a Lucas-Stokey economy in which for each period $t \geq 0$, the government promises to pay the same state-contingent amount $\tilde{b}$ in each state tomorrow.

This insight tells us to find a $b_0$ and other fundamentals for the Lucas-Stokey [LS83] model that make the Ramsey planner want to borrow the same value $\tilde{b}$ next period for all states and all dates.

We accomplish this by using various equations for the Lucas-Stokey [LS83] model presented in *optimal taxation with state-contingent debt*.

We use the following steps:
Step 1: Pick an initial $\Phi$.

Step 2: Given that $\Phi$, jointly solve two versions of equation (9.140) for $c(s)$, $s = 1, 2$ associated with the two values for $g(s)$, $s = 1, 2$

Step 3: Solve the following equation for $\bar{x}$

$$\bar{x} = (I - \beta \Pi)^{-1} [\bar{u}_c(\bar{n} - \bar{g}) - \bar{u}_l \bar{n}]$$  \hspace{1cm} (9.142)

Step 4: After solving for $\bar{x}$, 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

$$\bar{b} = \frac{\bar{x}}{u_c}$$  \hspace{1cm} (9.143)

Step 5: Compute $J(\Phi) = (b(1) - b(2))^2$

Step 6: Put steps 2 through 6 in a function minimizer and find a $\Phi$ that minimizes $J(\Phi)$

Step 7: At the value of $\Phi$ and the value of $\bar{b}$ that emerged from step 6, solve equations (9.141) and (9.139) jointly for $c_0, b_0$

### 9.6.6 Code for reverse engineering

Here is code to do the calculations for us

```python
from scipy.optimize import fsolve, fmin
u = CRRAutility()

def min_\Phi(\Phi):
    g1, g2 = u.G  # Government spending in s=0 and s=1

    # Solve \Phi(c)
    def equations(unknowns, \Phi):
        c1, c2 = unknowns
        # First argument of .Uc and second argument of .Un are redundant

        # Set up simultaneous equations
        eq = lambda c, g: (1 + \Phi) * (u.Uc(c, 1) - u.Un(1, c + g)) + \Phi * ((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=(\Phi))
```

9.6. Fluctuating Interest Rates Deliver Fiscal Insurance
We obtain

Optimization terminated successfully.
Current function value: 0.000000
Iterations: 24
Function evaluations: 48

To recover and print out \( \tilde{b} \)

\[
\tilde{b} = b[0] \\
\tilde{b} = \tilde{b} \\
\tilde{b} = -1.0757576567504166
\]

To complete the reverse engineering exercise by jointly determining \( c_0, b_0 \), we set up a function that returns two simultaneous equations

\[
def \text{solve}_\text{cb}(\text{unknowns}, \Phi, b_{\text{bar}}, s=1):
\]

\[
c_0, b_0 = \text{unknowns}
\]

\[
g_0 = u.G[s-1]
\]

\[
R_0 = u.\beta * u.\pi[s] @ [u.Uc(c1, 1) / u.Uc(c0, 1), u.Uc(c2, 1) / u.Uc(c0, 1)]
\]

\[
R_0 = 1 / R_0
\]

\[
\tau_0 = 1 + u.Un(1, c0 + g0) / u.Uc(c0, 1)
\]

\[
eq1 = \tau_0 * (c0 + g0) + b_{\text{bar}} / R_0 - b_0 - g0
\]

\[
eq2 = (1 + \Phi) * (u.Uc(c0, 1) + u.Un(1, c0 + g0)) + \Phi * (c0 + u.Ucc(c0, 1) + (c0 + g0) + u.Unn(1, c0 + g0)) - \Phi * u.Ucc(c0, 1) + b0
\]

\[
return np.array([eq1, eq2])
\]
To solve the equations for $c_0, b_0$, we use SciPy's `fsolve` function:

```
c0, b0 = fsolve(solve_cb, [1., -1.], args=(phi_star, b[0], 1), xtol=1.0e-12)
c0, b0
```

```
(0.9344994030900681, -1.038698407551764)
```

Thus, we have reverse engineered an initial $b_0 = -1.038698407551764$ that ought to render the AMSS measurability constraints slack.

### 9.6.7 Short simulation for reverse-engineered: initial debt

The following graph shows simulations of outcomes for both a Lucas-Stokey economy and for an AMSS economy starting from initial government debt equal to $b_0 = -1.038698407551764$

These graphs report outcomes for both the Lucas-Stokey economy with complete markets and the AMSS economy with one-period risk-free debt only.

```python
import matplotlib.pyplot as plt

_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, _grid, tol_diff=1e-10, tol=1e-12)

T = 20
sHist = np.array([0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 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)


# Government spending paths
sim_seq[4] = log_example.G[sHist]

# Output paths

fig, axes = plt.subplots(3, 2, figsize=(14, 10))

for ax, title, seq, bel in zip(axes.flatten(), titles, sim_seq, sim_bel):
```

9.6. Fluctuating Interest Rates Deliver Fiscal Insurance
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.

**9.6.8 Long simulation**

The following graph shows the par value of government debt and the flat rate tax on labor income for a long simulation for our sample economy.

For the **same** realization of a government expenditure path, the graph reports outcomes for two economies.
• the gray lines are for the Lucas-Stokey economy with complete markets
• the blue lines are for the AMSS economy with risk-free one-period debt only

For both economies, initial government debt due at time 0 is \( b_0 = .5 \)

For the Lucas-Stokey complete markets economy, the government debt plotted is \( b_{t+1}(s_{t+1}) \)

• Notice that this is a time-invariant function of the Markov state from the beginning

For the AMSS incomplete markets economy, the government debt plotted is \( b_{t+1}(s^t) \)

• Notice that this is a martingale-like random process that eventually seems to converge to a constant \( \bar{b} \approx -1.07 \)

• Notice that the limiting value \( \bar{b} < 0 \) so that asymptotically the government makes a constant level of risk-free loans to the public

• In the simulation displayed as well as other simulations we have run, the par value of government debt converges to about 1.07 after between 1400 to 2000 periods

For the AMSS incomplete markets economy, the marginal tax rate on labor income \( \tau_t \) converges to a constant

• labor supply and output each converge to time-invariant functions of the Markov state

```python
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()
```
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

9.6.9 BEGS approximations of limiting debt and convergence rate

It is useful to link the outcome of our reverse engineering exercise to limiting approximations constructed by [BEGS17]

[BEGS17] used a slightly different notation to represent a generalization of the AMSS model.
Well 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 $B_t, B_t, R_t, X_t$ that are related to our notation by

\[
R_t = \frac{u_{c,t}}{u_{c,t-1}} R_{t-1} = \frac{u_{c,t}}{\beta E_{t-1} u_{c,t}} \\
B_t = \frac{b_{t+1}(s^t)}{R_t(s^t)} \\
b_t(s^{t-1}) = R_{t-1} B_{t-1} \\
B_t = u_{c,t} B_t = (\beta E_t u_{c,t+1}) b_{t+1}(s^t) \\
X_t = u_{c,t}[g_t - \tau_t n_t]
\]

In terms of their notation, equation (44) of \cite{BEGS17} expresses the time $t$ state $s$ government budget constraint as

\[
B(s) = R_\tau(s, s_-) B_- + X_\tau(s)
\]  

(9.144)

where the dependence on $\tau$ is to remind us that these objects depend on the tax rate and $s_-$ is last periods Markov state

BEGS interpret random variations in the right side of (9.144) as a measure of fiscal risk composed of

- interest-rate-driven fluctuations in time $t$ effective payments due on the government portfolio, namely, $R_\tau(s, s_-) B_-$, and
- fluctuations in the effective government deficit $X_t$

**Asymptotic mean**

BEGS give conditions under which the ergodic mean of $B_t$ is

\[
B^* = - \frac{\text{cov}^\infty(R, X)}{\text{var}^\infty(R)}
\]  

(9.145)

where the superscript $\infty$ denotes a moment taken with respect to an ergodic distribution

Formula (9.145) presents $B^*$ as a regression coefficient of $X_t$ on $R_t$ in the ergodic distribution

This regression coefficient emerges as the minimizer for a variance-minimization problem:

\[
B^* = \arg\min_B \text{var}(RB + X)
\]  

(9.146)

The minimand in criterion (9.146) is the measure of fiscal risk associated with a given tax-debt policy that appears on the right side of equation (9.144)

Expressing formula (9.145) in terms of our notation tells us that $\bar{b}$ should approximately equal

\[
\bar{b} = \frac{B^*}{\beta E_t u_{c,t+1}}
\]  

(9.147)
Rate of convergence

BEGS also derive the following approximation to the rate of convergence to $B^*$ from an arbitrary initial condition

$$\frac{E_t(B_{t+1} - B^*)}{(B_t - B^*)} \approx \frac{1}{1 + \beta^2 \text{var}(\mathcal{R})} \quad (9.148)$$

(See the equation above equation (47) in [BEGS17])

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 .5, .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 (9.147).

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 (9.147).

After that, we move on to compute $B^*$ in formula (9.145).

We also evaluate the BEGS criterion (9.144) at the limiting value $B^*$

$$J(B^*) = \text{var}(\mathcal{R})(B^*)^2 + 2B^* \text{cov}(\mathcal{R}, \mathcal{X}) + \text{var}(\mathcal{X}) \quad (9.149)$$

Here are some functions that we use to compute key objects that we want.
```python
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 lets form the two random variables $R, X$ appearing in the BEGS approximating formulas

```python
u = CRRAutility()
s = 0
c = [0.94058082425584, 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 lets compute the ingredient of the approximating limit and the approximating rate of convergence

```python
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
```

Print out $\hat{b}$ and $\overline{b}$

```python
bhat, b_bar
```

```
(-1.0757585378303758, -1.0757576567504166)
```

So we have

```python
bhat - b_bar
```

9.6. Fluctuating Interest Rates Deliver Fiscal Insurance 1245
These outcomes show that $\hat{b}$ does a remarkably good job of approximating $\hat{b}$

Next, let's compute the BEGS fiscal criterion that $\hat{b}$ is minimizing

$$J_{\text{min}} = \text{variance}(R) \ast \text{bstar} \ast 2 + 2 \ast \text{bstar} \ast \text{covariance}(R, X) + \text{variance}(X)$$

This is machine zero, a verification that $\hat{b}$ succeeds in minimizing the nonnegative fiscal cost criterion $J(B^*)$ defined in BEGS and in equation (9.149) above

Let's push our luck and compute the mean reversion speed in the formula above equation (47) in [BEGS17]

$$\text{den2} = 1 + (u \ast \text{bstar} \ast 2) \ast \text{variance}(R)$$

$$\text{speedrever} = 1 / \text{den2}$$

$$\text{print}(\text{f'\text{Mean reversion speed = } (\text{speedrever})'}$$

Mean reversion speed = 0.9974715478249827

Now let's compute the implied mean time to get to within .01 of the limit

$$\text{ttime} = \text{np.log(.01) / np.log(speedrever)}$$

$$\text{print}(\text{f'\text{Time to get within .01 of limit = } (\text{ttime})'}$$

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

### 9.7 Fiscal Risk and Government Debt

This lecture studies government debt in an AMSS economy [AMSS02] 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 [BEGS17] (BEGS)

We study an [AMSS02] 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, [BEGS17] (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 [BEGS17] section III.D carry typos that we correct below

### 9.7.1 The economy

As in Optimal Taxation without State-Contingent Debt and Optimal Taxation with State-Contingent Debt, we assume that the representative agent has utility function

\[ u(c, n) = \frac{c^{1-\sigma}}{1 - \sigma} - \frac{n^{1+\gamma}}{1 + \gamma} \]

We work directly with labor supply instead of leisure

We assume that

\[ c_t + g_t = n_t \]

The Markov state \( s_t \) takes three values, namely, 0, 1, 2

The initial Markov state is 0

The Markov transition matrix is \((1/3)I\) where \( I \) is a 3 \times 3 identity matrix, so the \( s_t \) process is i.i.d.

Government expenditures \( g(s) \) equal .1 in Markov state 0, .2 in Markov state 1, and .3 in Markov state 2

We set preference parameters

\[ \beta = .9 \]
\[ \sigma = 2 \]
\[ \gamma = 2 \]

The following Python code sets up the economy
import numpy as np

class CRRAutility:
    def __init__(self, 
        β=0.9, 
        σ=2, 
        γ=2, 
        π=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**(1 - self.σ)

    def Ucc(self, c, n):
        return -self.σ * c**(1 - self.σ - 1)

    def Un(self, c, n):
        return -n**self.γ

    def Unn(self, c, n):
        return -self.γ * n**(self.γ - 1)

First and second moments

Well 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

def mean(x, s):
    '''Returns mean for x given initial state'''
    x = np.array(x)
    return x @ u.π[s]

def variance(x, s):
9.7.2 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

```python
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 \( \mu \).
    '''
    def __init__(self, model):
        # Initialize from model object attributes
        self.\beta, self.\pi, self.G = model.\beta, model.\pi, model.G
        self.mc, self.\Theta = MarkovChain(self.\pi), model.\Theta
        self.S = len(model.\pi)  # 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, \Theta, G = self.S, self.\Theta, self.G
        Uc, Un = model.Uc, model.Un

        def res(z):
            c = z[:S]
            n = z[S:]
            return np.hstack([\Theta * Uc(c, n) + Un(c, n), \Theta * n - c - G])
```

9.7. Fiscal Risk and Government Debt
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.zFB = np.hstack([self.cFB, self.nFB, self.zFB])

def timel_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:]
        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)) + \ # FOC of c
                          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):
\[ x_{\text{prime}} = \text{self}.\text{time1\_allocation}(\mu)[2] \]

\[ \text{return np.hstack([Uc(c, n) \ast (c - B_) + Un(c, n) \ast n + \beta \ast \pi[s_0],} \]

\[ \Theta x_{\text{prime}}, \]

\[ Uc(c, n) - \mu \ast (Ucc(c, n) \ast (c - B_) + Uc(c, n)) - \Xi, \]

\[ Un(c, n) - \mu \ast (Unn(c, n) \ast n + Un(c, n)) + \Theta[s_0] \ast \Xi, \]

\[ (\Theta \ast n - c - G)[s_0]) \)

# Find root
res = root(FOC, np.array([0, self.cFB[s_0], self.nFB[s_0], self.\XiFB[s_0]]))
if not res.success:
    raise Exception('Could not find time 0 LS allocation.')

return res.x

def timel\_value(self, \mu):
    '''
    Find the value associated with multiplier \mu
    '''
    c, n, x, \Xi = self.timel\_allocation(\mu)
    U = self.model.U(c, n)
    V = np.linalg.solve(np.eye(self.S) - self.\beta \ast self.\pi, 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.\Theta \ast Uc)

def simulate(self, B_, s_0, T, sHist=None):
    '''
    Simulates planners policies for T periods
    '''
    model, \pi, \beta = self.model, self.\pi, self.\beta
    Uc = model.Uc

    if sHist is None:
        sHist = self.mc.simulate(T, s_0)

    cHist, nHist, Bhist, THist, \muHist = np.zeros((5, T))
    RHist = np.zeros(T - 1)

    # Time 0
    \mu, cHist[0], nHist[0], _ = self.time0\_allocation(B_, s_0)
    THist[0] = self.T(cHist[0], nHist[0])[s_0]
    Bhist[0] = B_
\[
\mu_{\text{Hist}[0]} = \mu
\]

# Time 1 onward
for t in range(1, T):
    c, n, x, \Xi = self.time1_allocation(\mu)
    T = self.T(c, n)
    u_c = Uc(c, n)
    s = sHist[t]
    Eu_c = \pi(sHist[t - 1]) \otimes u_c
    cHist[t], nHist[t], Bhist[t], THist[t] = c[s], n[s], x[s] / \backslash
    u_c[s], T[s]
    RHist[t - 1] = Uc(cHist[t - 1], nHist[t - 1]) / (\beta \ast Eu_c)
    \mu_{\text{Hist}[t]} = \mu

return np.array([cHist, nHist, Bhist, THist, sHist, \mu_{\text{Hist}}, RHist])

from scipy.optimize import fmin_slsqp

class RecursiveAllocationAMSS:
    def __init__(self, model, \mu_{\text{grid}}, tol_diff=1e-4, tol=1e-4):
        self.\beta, self.\pi, self.G = model.\beta, model.\pi, model.G
        self.mc, self.S = MarkovChain(self.\pi), len(model.\pi)  # Number of states

        self.\Theta, self.model, self.\mu_{\text{grid}} = model.\Theta, model, \mu_{\text{grid}}
        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 \mu_{\text{grid0}}
        '''
        model, \mu_{\text{grid0}} = self.model, self.\mu_{\text{grid}}
        \pi = model.\pi
        S = len(model.\pi)

        # 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(\_\_\_, s_):
            c, n, V = PP.time1_value(\_\_\_)
            return c, n, \pi(s_) \otimes x, \pi(s_) \otimes V
        cf, nf, xgrid, Vf, xprimef = [], [], [], [], []
        for s_ in range(S):
            c, n, x, V = zip(*map(lambda \mu: incomplete_allocation(\mu, s_), \mu_{\text{grid0}}))

        1252 Chapter 9. Dynamic Programming Squared
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))
np.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.\* \* 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, \ = self.model, self.\pi
    Uc = model.Uc
    cf, nf, xprimef, Tf = self.policies

    if sHist is None:
        sHist = simulate_markov(\, s_0, T)

    cHist, nHist, Bhist, xHist, THist, \Hist, \Hist, \Hist = np.zeros((7, T))

    # time 0
    cHist[0], nHist[0], xHist[0], THist[0] = self.time0_allocation(B_, s_0)
    BHist[0] = B_
    \Hist[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 = \pi[s_, :] \* u_c

        \Hist[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, \muHist, sHist, xHist])

class BellmanEquation:
    '''
    Bellman equation for the continuation of the Lucas-Stokey Problem
    '''

    def __init__(self, model, xgrid, policies0, tol, maxiter=1000):
        self.\beta, self.\pi, self.G = model.\beta, model.\pi, model.G
        self.S = len(model.\pi)  # Number of states
        self.\Theta, self.model, self.tol = model.\Theta, 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, \Theta, Uc, Un, G = self.S, self.\Theta, model.Uc, model.Un, self.G

        def res(z):
            c = z[:S]
            n = z[S:]
            return np.hstack([\Theta * Uc(c, n) + Un(c, n), \Theta * 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.\beta * self.\pi, IFB)
self.zFB = {}
for s in range(S):
    self.zFB[s] = np.hstack([self.cFB[s], self.nFB[s], self.pi[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
    '''
    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 -(s_ @ (U(c, n) + beta * 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 = pi[s_] @ u_c
    return np.hstack([x * u_c / Eu_c - u_c * (c - T) - Un(c, n) * n - beta * xprime,
                      theta * 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])

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]])
Next, we show code that we use to generate a very long simulation starting from initial government debt equal to $-0.5$.

Here is a graph of a long simulation of 102000 periods:

```python
import matplotlib.pyplot as plt

_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()
```
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 \[BEGS17\] 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 $\overline{B^* \Eu}$, associated with the sample (presumably) from the ergodic where $B^*$ is the regression coefficient described below; the red vertical line denotes an approximation by [BEGS17] 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:

```python
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()
9.7 Fiscal Risk and Government Debt

Graphs showing the relationship between Government Debt and Tax Rate under different market conditions (Complete Markets vs. Incomplete Markets). The graphs demonstrate fluctuations in debt and tax levels over time.
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

### 9.7.3 Asymptotic mean and rate of convergence

We apply results of [BEGS17] to interpret

• the mean of the ergodic distribution of government debt

• the rate of convergence to the ergodic distribution from an arbitrary initial government debt

We begin by computing objects required by the theory of section III.i of [BEGS17]

As in *Fiscal Insurance via Fluctuating Interest Rates*, we recall that [BEGS17] used a particular notation to represent what we can regard as a generalization of the AMSS model

We introduce some of the [BEGS17] 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 $B_t, B_t, R_t, X_t$ that are related to notation that we used in earlier lectures by

\[
\begin{align*}
R_t &= \frac{u_{c,t}}{u_{c,t-1}} R_t^{t-1} = \frac{u_{c,t}}{\beta E_t u_{c,t}} \\
B_t &= \frac{b_{t+1}(s^t)}{R_t(s^t)} \\
b_t(s^{t-1}) &= R_{t-1} B_{t-1} \\
B_t &= u_{c,t} B_t = (\beta E_t u_{c,t+1}) b_{t+1}(s^t) \\
X_t &= u_{c,t} [g_t - \tau_t n_t]
\end{align*}
\]

[BEGS17] call $X_t$ the effective government deficit, and $B_t$ the effective government debt

Equation (44) of [BEGS17] expresses the time $t$ state $s$ government budget constraint as

\[
B(s) = R_{\tau}(s, s_-) B_- + X_{\tau}(s)
\]  

(9.150)

where the dependence on $\tau$ is to remind us that these objects depend on the tax rate; $s_-$ is last periods Markov state

BEGS interpret random variations in the right side of (9.150) as fiscal risks generated by

• interest-rate-driven fluctuations in time $t$ effective payments due on the government portfolio, namely, $R_{\tau}(s, s_-) B_-$, and

• fluctuations in the effective government deficit $X_t$
**Asymptotic mean**

BEGS give conditions under which the ergodic mean of $B_t$ approximately satisfies the equation

$$B^* = -\frac{\text{cov}^\infty(R_t, X_t)}{\text{var}^\infty(R_t)}$$  \hspace{1cm} (9.151)

where the superscript $\infty$ denotes a moment taken with respect to an ergodic distribution.

Formula (9.151) represents $B^*$ as a regression coefficient of $X_t$ on $R_t$ in the ergodic distribution.

Regression coefficient $B^*$ solves a variance-minimization problem:

$$B^* = \text{argmin}_B \text{var}^\infty(RB + X)$$ \hspace{1cm} (9.152)

The minimand in criterion (9.152) measures fiscal risk associated with a given tax-debt policy that appears on the right side of equation (9.150).

Expressing formula (9.151) 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

$$\hat{b} = \frac{B^*}{\beta E(E_t u_{c,t+1})} = \frac{B^*}{\beta E(u_{c,t+1})}$$ \hspace{1cm} (9.153)

where mathematical expectations are taken with respect to the ergodic distribution.

**Rate of convergence**

BEGS also derive the following approximation to the rate of convergence to $B^*$ from an arbitrary initial condition

$$\frac{E_t(B_{t+1} - B^*)}{(B_t - B^*)} \approx \frac{1}{1 + \beta^2 \text{var}^\infty(R)}$$ \hspace{1cm} (9.154)

(See the equation above equation (47) in [BEGS17])

**More advanced material**

The remainder of this lecture is about technical material based on formulas from [BEGS17].

The topic is interpreting and extending formula (9.152) for the ergodic mean $B^*$.
Chicken and egg

Attributes of the ergodic distribution for $B_t$ appear on the right side of formula (9.152) for the ergodic mean $B^*$

Thus, formula (9.152) 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 (9.152)

So the primary use of equation (9.152) 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 $B$ 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 $B$ agrees with the mean of that distribution

Approximating $B^*$

[BEGS17] propose an approximation to $B^*$ that can be computed without first knowing the ergodic distribution

To construct the BEGS approximation to $B^*$, we just follow steps set forth on pages 648 - 650 of section III.D of [BEGS17]

- notation in BEGS might be confusing at first sight, so it is important to stare and digest before computing
- there are also some sign errors in the [BEGS17] text that we want to correct

Here is a step-by-step description of the [BEGS17] approximation procedure

**Step by step**

**Step 1:** For a given $\tau$ we compute a vector of values $c_\tau(s), s = 1, 2, \ldots, S$ that satisfy

\[
(1 - \tau)c_\tau(s)^{-\sigma} - (c_\tau(s) + g(s))^\gamma = 0
\]

This is a nonlinear equation to be solved for $c_\tau(s), s = 1, \ldots, S$

$S = 3$ in our case, but we will write code for a general integer $S$

**Typo alert:** Please note that there is a sign error in equation (42) of [BEGS17] – it should be a minus rather than a plus in the middle

- We have made the appropriate correction in the above equation

**Step 2:** Knowing $c_\tau(s), s = 1, \ldots, S$ for a given $\tau$, we want to compute the random variables

\[
R_\tau(s) = \frac{c_\tau(s)^{-\sigma}}{\beta \sum_{s' = 1}^{S} c_\tau(s')^{-\sigma} \pi(s')}
\]
and
\[ X_\tau(s) = (c_\tau(s) + g(s))^{1+\gamma} - c_\tau(s)^{1-\sigma} \]
each for \( s = 1, \ldots, S \)

BEGS call \( R_\tau(s) \) the **effective return** on risk-free debt and they call \( X_\tau(s) \) the **effective government deficit**

**Step 3:** With the preceding objects in hand, for a given \( B \), we seek a \( \tau \) that satisfies
\[
B = -\frac{\beta}{1 - \beta} E X_\tau = -\frac{\beta}{1 - \beta} \sum_s X_\tau(s) \tau(s)
\]
This equation says that at a constant discount factor \( \beta \), equivalent government debt \( B \) equals the present value of the mean effective government surplus

**Typo alert:** there is a sign error in equation (46) of [BEGS17] – the left side should be multiplied by \(-1\)

- We have made this correction in the above equation

For a given \( B \), let a \( \tau \) that solves the above equation be called \( \tau(B) \)

Well use a Python root solver to finds a \( \tau \) that this equation for a given \( B \)

Well use this function to induce a function \( \tau(B) \)

**Step 4:** With a Python program that computes \( \tau(B) \) in hand, next we write a Python function to compute the random variable
\[
J(B)(s) = R_\tau(B)(s)B + X_\tau(B)(s), \quad s = 1, \ldots, S
\]

**Step 5:** Now that we have a machine to compute the random variable \( J(B)(s), s = 1, \ldots, S, \) via a composition of Python functions, we can use the population variance function that we defined in the code above to construct a function \( \text{var}(J(B)) \)

We put \( \text{var}(J(B)) \) into a function minimizer and compute
\[
B^* = \text{argmin}_B \text{var}(J(B))
\]

**Step 6:** Next we take the minimizer \( B^* \) and the Python functions for computing means and variances and compute
\[
\text{rate} = \frac{1}{1 + \beta^2 \text{var}(R_\tau(B^*))}
\]

Ultimate outputs of this string of calculations are two scalars
\[
(B^*, \text{rate})
\]

**Step 7:** Compute the divisor
\[
div = \beta Eu_{c,t+1}
\]
and then compute the mean of the par value of government debt in the AMSS model
\[
\hat{b} = \frac{B^*}{div}
\]
In the two-Markov-state AMSS economy in *Fiscal Insurance via Fluctuating Interest Rates*, $E_t u_{c,t+1} = \mathbb{E} u_{c,t+1}$ 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

**Execution**

Now let's move on to compute things step by step

**Step 1**

```python
u = CRRAutility(\(\pi=(1 / 3) * np.\text{ones}(3, 3)\)),
G=np.array([0.1, 0.2, 0.3]),
Theta=np.\text{ones}(3))

\(\tau = 0.05\) # Initial guess of \(\tau\) (to displays calcs along the way)
S = len(u.G) # Number of states

\text{def} \text{solve}_c(c, \tau, u):
    \text{return} (1 - \tau) * c**(-u.\sigma) - (c + u.G)**u.\gamma

c = \text{root}(\text{solve}_c, np.\text{ones}(S), \text{args}=(\tau, u)).x # \text{.x returns the result from }\text{root}
```

array([0.93852387, 0.89231015, 0.84858872])

\text{root}(\text{solve}_c, np.\text{ones}(S), \text{args}=(\tau, u))

```
fjac: array([[-0.99990816, -0.00495351, -0.01261467],
              [-0.00515633, 0.99985715, 0.01609659],
              [-0.01253313, -0.01616015, 0.99979086])]
fun: array([ 5.61814373e-10, -4.76900741e-10,  1.17474919e-11])
message: 'The solution converged.'
```
Step 2

\[ n = c + u.G \quad \# \text{compute labor supply} \]

Note about code

Remember that in our code \( \pi \) is a 3 \( \times \) 3 transition matrix

But because we are studying an i.i.d. case, \( \pi \) has identical rows and we only need to compute objects for one row of \( \pi \)

This explains why at some places below we set \( s = 0 \) just to pick off the first row of \( \pi \) in the calculations

Code

First, let's compute \( R \) and \( X \) according to our formulas

```python
def compute_R_X(\tau, u, s):
    c = root(solve_c, np.ones(S), args=(\tau, u)).x \quad \# \text{Solve for vector of } c's
    div = u.\beta * (u.Uc(c[0], n[0]) * u.\pi[s, 0] + u.Uc(c[1], n[1]) * u.\pi[s, 1]) + u.Uc(c[2], n[2]) * u.\pi[s, 2])
    R = c**(-u.\sigma) / (div)
    X = (c + u.G)**(1 + u.\gamma) - c**(1 - u.\sigma)
    return R, X
```

\[ c**(-u.\sigma) @ u.\pi \]

```plaintext
array([1.25997521, 1.25997521, 1.25997521])
```

\[ u.\pi \]

```plaintext
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 well set \( s = 0 \) and just pick off expectations associated with the first row of \( \pi \)
s = 0
R, X = compute_R_X(τ, u, s)

Lets look at the random variables $R, X$

$R$

array([1.00116313, 1.10755123, 1.22461897])

mean(R, s)

1.1111111111111112

$X$

array([0.05457803, 0.18259396, 0.33685546])

mean(X, s)

0.19134248445303795

$X @ u.π$

array([0.19134248, 0.19134248, 0.19134248])

**Step 3**

```python
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

Lets try out our method computing $τ$

s = 0
B = 1.0
τ = root(solve_τ, .1, args=(B, u, s)).x[0]  # Very sensitive to starting value

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

```python
def min_J(B, u, s):
    \tau = \text{root}(\text{solve}_\tau, .5, \text{args}=(B, u, s)).x[0]  # very sensitive to initial value of \tau
    R, X = \text{compute}_R_X(\tau, u, s)
    return \text{variance}(R * B + X, s)
```

```
min_J(B, u, s)
```

```
0.035564405653720765
```

Step 6

```python
from scipy.optimize import minimize

B_star = minimize(min_J, .5, args=(u, s)).x[0]
B_star
```

```
-1.199483167941158
```

```python
n = c + u.G  # compute labor supply

div = u.\beta * (u.Uc(c[0], n[0]) * u.\pi[s, 0] + u.Uc(c[1], n[1]) * u.\pi[s, 1] + u.Uc(c[2], n[2]) * u.\pi[s, 2])

B_hat = B_star/div
B_hat
```

```
-1.0577661126390971
```

```python
\tau_star = \text{root}(\text{solve}_\tau, 0.05, \text{args}=(B_star, u, s)).x[0]
\tau_star
```

```
0.09572916798461703
```

```python
R_star, X_star = \text{compute}_R_X(\tau_star, u, s)
R_star, X_star
```

```
(array([0.9998398 , 1.10746593, 1.2260276 ]),
array([0.0020272 , 0.12464752, 0.27315299]))
```

```python
rate = 1 / (1 + u.\beta**2 * \text{variance}(R_star, s))
rate
```

9.7. Fiscal Risk and Government Debt 1271
9.8 Competitive Equilibria of Chang Model

Co-author: Sebastian Graves

9.8.1 Overview

This lecture describes how Chang [Cha98] analyzed competitive equilibria and a best competitive equilibrium called a Ramsey plan.

He did this by

• characterizing a competitive equilibrium recursively in a way also employed in dynamic Stackelberg problems, the Calvo model, and history dependent public policies lecture to pose Stackelberg problems in linear economies, and then

• appropriately adapting an argument of Abreu, Pearce, and Stachetti [APS90] to describe key features of the set of competitive equilibria

Roberto Chang [Cha98] chose a model of Calvo [Cal78] as a simple structure that conveys ideas that apply more broadly

A textbook version of Changs model appears in chapter 25 of [LS18]

This lecture and Credible Government Policies in Chang Model can be viewed as more sophisticated and complete treatments of the topics discussed in Ramsey plans, time inconsistency, sustainable plans
Both this lecture and *Credible Government Policies in Chang Model* make extensive use of an idea to which we apply the nickname **dynamic programming squared**

In dynamic programming squared problems there are typically two interrelated Bellman equations

- A Bellman equation for a set of agents or followers with value or value function $v_a$
- A Bellman equation for a principal or Ramsey planner or Stackelberg leader with value or value function $v_p$ in which $v_a$ appears as an argument

We encountered problems with this structure in *dynamic Stackelberg problems, optimal taxation with state-contingent debt*, and other lectures

**The setting**

First, we introduce some notation

For a sequence of scalars $\bar{z} \equiv \{z_t\}_{t=0}^\infty$, let $\bar{z}^t = (z_0, \ldots, z_t), \bar{z}_t = (z_t, z_{t+1}, \ldots)$

An infinitely lived representative agent and an infinitely lived government exist at dates $t = 0, 1,\ldots$

The objects in play are

- an initial quantity $M_{-1}$ of nominal money holdings
- a sequence of inverse money growth rates $\bar{h}$ and an associated sequence of nominal money holdings $\bar{M}$
- a sequence of values of money $\bar{q}$
- a sequence of real money holdings $\bar{m}$
- a sequence of total tax collections $\bar{x}$
- a sequence of per capita rates of consumption $\bar{c}$
- a sequence of per capita incomes $\bar{y}$

A benevolent government chooses sequences $(\bar{M}, \bar{h}, \bar{x})$ 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 $(\bar{c}, \bar{m})$ of consumption and real balances

In competitive equilibrium, the price of money sequence $\bar{q}$ clears markets, thereby reconciling decisions of the government and the representative household

Chang adopts a version of a model that [*Cal78*] designed to exhibit time-inconsistency of a Ramsey policy in a simple and transparent setting

By influencing the representative households 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 households anticipations of the governments future actions affect the households 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 $t \geq 0$.

### 9.8.2 Setting

#### The households problem

A representative household faces a nonnegative value of money sequence $\bar{q}$ and sequences $\bar{y}, \bar{x}$ of income and total tax collections, respectively.

The household chooses nonnegative sequences $\bar{c}, \bar{M}$ of consumption and nominal balances, respectively, to maximize

$$
\sum_{t=0}^{\infty} \beta^t [u(c_t) + v(q_t M_t)]
$$

subject to

$$
q_t M_t \leq y_t + q_t M_{t-1} - c_t - x_t
$$

and

$$
q_t M_t \leq \bar{m}
$$

Here $q_t$ is the reciprocal of the price level at $t$, which we can also call the *value of money*.

Chang [Cha98] assumes that

- $u : \mathbb{R}_+ \rightarrow \mathbb{R}$ is twice continuously differentiable, strictly concave, and strictly increasing;
- $v : \mathbb{R}_+ \rightarrow \mathbb{R}$ is twice continuously differentiable and strictly concave;
- $u'(c)_{c \rightarrow 0} = \lim_{m \rightarrow 0} v'(m) = +\infty$;
- 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 (9.156) is the households 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 (9.157) imposes an exogenous upper bound $\bar{m}$ on the households choice of real balances, where $\bar{m} \geq m^f$. 

1274 Chapter 9. Dynamic Programming Squared
Government

The government chooses a sequence of inverse money growth rates with time $t$ component $h_t \in \Pi \equiv [\pi, \overline{\pi}]$, where $0 < \pi < 1 < \frac{1}{\beta} < \overline{\pi}$

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

$$-x_t = m_t(1 - h_t) \quad (9.158)$$

The restrictions $m_t \in [0, \bar{m}]$ and $h_t \in \Pi$ evidently imply that $x_t \in X \equiv [(\pi - 1)m_t, (\pi - 1)\bar{m}]$

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:

$$y_t = f(x_t), \quad (9.159)$$

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

Calvos and Changs 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 $(9.155)$

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 $y$, the household chooses $M_t$ and therefore $m_t$ because $m_t = q_tM_t$;
- 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

9.8. Competitive Equilibria of Chang Model
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*.

### Households problem

Given $M_{t-1}$ and $\{q_t\}_{t=0}^{\infty}$, the households problem is

$$\mathcal{L} = \max_{\bar{c}, \bar{M}} \min_{\lambda, \bar{\mu}} \sum_{t=0}^{\infty} \beta^t \{ u(c_t) + v(M_t q_t) + \lambda [y_t - c_t - x_t + q_t M_{t-1} - q_t M_t] \\
+ \mu [\bar{m} - q_t M_t] \}$$

First-order conditions with respect to $c_t$ and $M_t$, respectively, are

$$u'(c_t) = \lambda_t$$
$$q_t [u'(c_t) - u'(M_t q_t)] \leq \beta u'(c_{t+1}) q_{t+1}, \quad \text{if} \quad M_t q_t < \bar{m}$$

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

$$m_t [u'(c_t) - u'(m_t)] \leq \beta u'(f(x_{t+1})) m_{t+1} h_{t+1}, \quad \text{if} \quad m_t < \bar{m} \quad (9.160)$$

Define the following key variable

$$\theta_{t+1} \equiv u'(f(x_{t+1})) m_{t+1} h_{t+1} \quad (9.161)$$

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 (9.161) shows that $\theta_{t+1}$ intermediates the influences of $(\bar{x}_{t+1}, \bar{m}_{t+1})$ on the households choice of real balances $m_t$.

By intermediates we mean that the future paths $(\bar{x}_{t+1}, \bar{m}_{t+1})$ influence $m_t$ entirely through their effects on the scalar $\theta_{t+1}$.

The observation that the one dimensional promised marginal utility of real balances $\theta_{t+1}$ functions in this way is an important step in constructing a class of competitive equilibria that have a recursive representation.

A closely related observation pervaded the analysis of Stackelberg plans in lecture *dynamic Stackelberg problems*. 

---

1276 Chapter 9. Dynamic Programming Squared
9.8.3 Competitive equilibrium

**Definition:**

- A government policy is a pair of sequences $(\vec{h}, \vec{x})$ where $h_t \in \Pi \forall t \geq 0$.
- A price system is a nonnegative value of money sequence $\vec{q}$.
- An allocation is a triple of nonnegative sequences $(\vec{c}, \vec{m}, \vec{y})$.

It is required that time $t$ components $(m_t, x_t, h_t) \in E$.

**Definition:**

Given $M_{-1}$, a government policy $(\vec{h}, \vec{x})$, price system $\vec{q}$, and allocation $(\vec{c}, \vec{m}, \vec{y})$ are said to be a competitive equilibrium if:

- $m_t = q_t M_t$ and $y_t = f(x_t)$
- The government budget constraint is satisfied.
- Given $\vec{q}, \vec{x}, \vec{y}, (\vec{c}, \vec{m})$ solves the households problem.

9.8.4 Inventory of objects in play

Chang constructs the following objects:

1. A set $\Omega$ of initial marginal utilities of money $\theta_0$.
   - Let $\Omega$ denote the set of initial promised marginal utilities of money $\theta_0$ associated with competitive equilibria.
   - Chang exploits the fact that a competitive equilibrium consists of a first period outcome $(h_0, m_0, x_0)$ and a continuation competitive equilibrium with marginal utility of money $\theta_1 \in \Omega$.

2. Competitive equilibria that have a recursive representation.
   - A competitive equilibrium with a recursive representation consists of an initial $\theta_0$ and a four-tuple of functions $(h, m, x, \Psi)$ mapping $\theta$ into this periods $(h, m, x)$ and next periods $\theta$, respectively.
   - A competitive equilibrium can be represented recursively by iterating on:

$$
\begin{align*}
  h_t &= h(\theta_t) \\
  m_t &= m(\theta_t) \\
  x_t &= x(\theta_t) \\
  \theta_{t+1} &= \Psi(\theta_t)
\end{align*}
$$

starting from $\theta_0$.

The range and domain of $\Psi(\cdot)$ are both $\Omega$.

3. A recursive representation of a Ramsey plan.
A recursive representation of a Ramsey plan is a recursive competitive equilibrium \( \theta_0, (h, m, x, \Psi) \) that, among all recursive competitive equilibria, maximizes \( \sum_{t=0}^{\infty} \beta^t [u(c_t) + v(q_t M_t)] \).

The Ramsey planner chooses \( \theta_0, (h, m, x, \Psi) \) from among the set of recursive competitive equilibria at time 0.

Iterations on the function \( \Psi \) determine subsequent \( \theta_t \)'s that summarize the aspects of the continuation competitive equilibria that influence the households decisions.

At time 0, the Ramsey planner commits to this implied sequence \( \{\theta_t\}_{t=0}^{\infty} \) and therefore to an associated sequence of continuation competitive equilibria.

4. A characterization of time-inconsistency of a Ramsey plan

Imagine that after a revolution at time \( t \geq 1 \), a new Ramsey planner is given the opportunity to ignore history and solve a brand new Ramsey plan.

This new planner would want to reset the \( \theta_t \) associated with the original Ramsey plan to \( \theta_0 \).

The incentive to reinitialize \( \theta_t \) associated with this revolution experiment indicates the time-inconsistency of the Ramsey plan.

By resetting \( \theta \) to \( \theta_0 \), the new planner avoids the costs at time \( t \) that the original Ramsey planner must pay to reap the beneficial effects that the original Ramsey plan for \( s \geq t \) had achieved via its influence on the households decisions for \( s = 0, \ldots, t-1 \).

### 9.8.5 Analysis

A competitive equilibrium is a triple of sequences \( (\vec{m}, \vec{x}, \vec{h}) \in E^\infty \) that satisfies (9.156), (9.157), and (9.160).

Chang works with a set of competitive equilibria defined as follows.

**Definition:** \( CE = \{ (\vec{m}, \vec{x}, \vec{h}) \in E^\infty \text{ such that (9.156), (9.157), and (9.160) are satisfied } \} \)

\( CE \) 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 [APS90] with insights of Kydland and Prescott [KP80a].

**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

\[ \max_{(\vec{m}, \vec{x}, \vec{h}) \in E^\infty} \sum_{t=0}^{\infty} \beta^t [u(c_t) + v(m_t)] \]

subject to restrictions (9.156), (9.157), and (9.160).
Evidently, associated with any competitive equilibrium \((m_0, x_0)\) is an implied value of \(\theta_0 = u'(f(x_0))(m_0 + x_0)\).

To bring out a recursive structure inherent in the Ramsey problem, Chang defines the set

\[
\Omega = \left\{ \theta \in \mathbb{R} \text{ such that } \theta = u'(f(x_0))(m_0 + x_0) \text{ for some } (\vec{m}, \vec{x}, \vec{h}) \in CE \right\}
\]

Equation (9.160) inherits from the households Euler equation for money holdings the property that the value of \(m_0\) consistent with the representative households choices depends on \((\vec{h}_1, \vec{m}_1)\).

This dependence is captured in the definition above by making \(\Omega\) be the set of first period values of \(\theta_0\) satisfying \(\theta_0 = u'(f(x_0))(m_0 + x_0)\) for first period component \((m_0, h_0)\) of competitive equilibrium sequences \((\vec{m}, \vec{x}, \vec{h})\).

Chang establishes that \(\Omega\) is a nonempty and compact subset of \(\mathbb{R}_+\).

Next Chang advances:

**Definition:** \(\Gamma(\theta) = \{(\vec{m}, \vec{x}, \vec{h}) \in CE | \theta = u'(f(x_0))(m_0 + x_0)\}\)

Thus, \(\Gamma(\theta)\) is the set of competitive equilibrium sequences \((\vec{m}, \vec{x}, \vec{h})\) whose first period components \((m_0, h_0)\) deliver the prescribed value \(\theta\) for first period marginal utility.

If we knew the sets \(\Omega, \Gamma(\theta)\), we could use the following two-step procedure to find at least the value of the Ramsey outcome to the representative household:

1. Find the indirect value function \(w(\theta)\) defined as

\[
w(\theta) = \max_{(\vec{m}, \vec{x}, \vec{h}) \in \Gamma(\theta)} \sum_{t=0}^{\infty} \beta^t [u(f(x_t)) + v(m_t)]
\]

2. Compute the value of the Ramsey outcome by solving \(\max_{\theta \in \Omega} w(\theta)\)

Thus, Chang states the following

**Proposition:**

\(w(\theta)\) satisfies the Bellman equation

\[
w(\theta) = \max_{x, m, h, \theta'} \left\{ u(f(x)) + v(m) + \beta w(\theta') \right\}
\]

where maximization is subject to

\[
(m, x, h) \in E \text{ and } \theta' \in \Omega
\]

and

\[
\theta = u'(f(x))(m + x)
\]
\[-x = m(1 - h) \quad (9.166)\]

and

\[m \cdot [u'(f(x)) - v'(m)] \leq \beta \theta', \quad \text{if } m < \bar{m} \quad (9.167)\]

Before we use this proposition to recover a recursive representation of the Ramsey plan, note that the proposition relies on knowing the set \(\Omega\)

To find \(\Omega\), Chang uses the insights of Kydland and Prescott [KP80a] together with a method based on the Abreu, Pearce, and Stacchetti [APS90] iteration to convergence on an operator \(B\) that maps continuation values into values

We want an operator that maps a continuation \(\theta\) into a current \(\theta\)

Chang lets \(Q\) be a nonempty, bounded subset of \(\mathbb{R}\)

Elements of the set \(Q\) are taken to be candidate values for continuation marginal utilities

Chang defines an operator

\[B(Q) = \theta \in \mathbb{R} \text{ such that there is } (m, x, h', \theta') \in E \times Q\]

such that (9.165), (9.166), and (9.167) hold

Thus, \(B(Q)\) is the set of first period \(\theta\)s attainable with \((m, x, h) \in E\) and some \(\theta' \in Q\)

**Proposition:**

1. \(Q \subset B(Q)\) implies \(B(Q) \subset \Omega\) (self-generation)
2. \(\Omega = B(\Omega)\) (factorization)

The proposition characterizes \(\Omega\) as the largest fixed point of \(B\)

It is easy to establish that \(B(Q)\) is a monotone operator

This property allows Chang to compute \(\Omega\) as the limit of iterations on \(B\) provided that iterations begin from a sufficiently large initial set

**Some useful notation**

Let \(\vec{h}^t = (h_0, h_1, \ldots, h_t)\) denote a history of inverse money creation rates with time \(t\) component \(h_t \in \Pi\)

A government strategy \(\sigma = \{\sigma_t\}_{t=0}^\infty\) is a \(\sigma_0 \in \Pi\) and for \(t \geq 1\) a sequence of functions \(\sigma_t : \Pi^{t-1} \rightarrow \Pi\)

Chang restricts the governments choice of strategies to the following space:

\[CE_\pi = \{\vec{h} \in \Pi^\infty : \text{ there is some } (\vec{m}, \vec{x}) \text{ such that } (\vec{m}, \vec{x}, \vec{h}) \in CE\}\]

In words, \(CE_\pi\) is the set of money growth sequences consistent with the existence of competitive equilibria
Chang observes that $CE_π$ is nonempty and compact

**Definition:** $σ$ is said to be admissible if for all $t ≥ 1$ 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

$$CE_π^0 = \{ h ∈ Π : \text{there is } \vec{h} ∈ CE_π \text{ with } h = h_0 \}.$$

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:** $CE_π^0 = \{ h ∈ Π : \text{there is } (m, θ') ∈ [0, \bar{m}] × Ω \text{ such that } μ'(f((h - 1)m) - v'(m)) ≤ βθ' \text{ with equality if } m < \bar{m} \}$

**Definition:** An allocation rule is a sequence of functions $\vec{α} = \{α_t\}_{t=0}^∞$ such that $α_t : Π^t → [0, \bar{m}] × X$

Thus, the time $t$ component of $α_t(h^t)$ 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 $h_t ∈ CE_π^0$, the continuations of $σ$ and $α$ after $(\vec{h}^{t-1}, h_t)$ induce a competitive equilibrium sequence

**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 $R$

Let $Z$ be a nonempty subset of $W × Ω$

Think of using pairs $(w', θ')$ drawn from $Z$ as candidate continuation value, $θ$ pairs

Define the operator

$$D(Z) = \{ (w, θ) : \text{there is } h ∈ CE_π^0 \text{ and a four-tuple } (m(h), x(h), w'(h), θ'(h)) ∈ [0, \bar{m}] × X × Z \}$$

### 9.8. Competitive Equilibria of Chang Model

1281
such that

\[ w = u(f(x(h))) + v(m(h)) + \beta w'(h) \quad (9.168) \]

\[ \theta = u'(f(x(h))) (m(h) + x(h)) \quad (9.169) \]

\[ x(h) = m(h)(h - 1) \quad (9.170) \]

\[ m(h)(u'(f(x(h))) - v'(m(h))) \leq \beta \theta'(h) \quad (9.171) \]

with equality if \( m(h) < \bar{m} \}

It is possible to establish

**Proposition:**

1. If \( Z \subset D(Z) \), then \( D(Z) \subset S \) (self-generation)
2. \( S = D(S) \) (factorization)

**Proposition:**

1. Monotonicity of \( D \): \( Z \subset Z' \) implies \( D(Z) \subset D(Z') \)
2. \( Z \) compact implies that \( D(Z) \) is compact

It can be shown that \( S \) is compact and that therefore there exists a \( (w, \theta) \) pair within this set that attains the highest possible value \( w \)

This \( (w, \theta) \) pair is 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

**9.8.6 Calculating all \( (w, \theta) \) pairs in \( CE \)**

Above we have defined the \( D(Z) \) operator as:

\[ D(Z) = \{(w, \theta) : \exists h \in CE^0 \text{ and } (m(h), x(h), w'(h), \theta'(h)) \in [0, \bar{m}] \times X \times Z \} \]

such that

\[ w = u(f(x(h))) + v(m(h)) + \beta w'(h) \]
\[
\theta = u'(f(x(h)))(m(h) + x(h)) \\
x(h) = m(h)(h - 1) \\
m(h)(u'(f(x(h))) - v'(m(h))) \leq \beta \theta'(h) \text{ (with equality if } m(h) < \bar{m})
\]

We noted that the set \( S \) can be found by iterating to convergence on \( D \), provided that we start with a sufficiently large initial set \( S_0 \).

Our implementation builds on ideas in this notebook.

To find \( S \) we use a numerical algorithm called the outer hyperplane approximation algorithm.

It was invented by Judd, Yeltekin, Conklin \([JYC03]\).

This algorithm constructs the smallest convex set that contains the fixed point of the \( D(S) \) operator.

Given that we are finding the smallest convex set that contains \( S \), we can represent it on a computer as the intersection of a finite number of half-spaces.

Let \( H \) be a set of subgradients, and \( C \) be a set of hyperplane levels.

We approximate \( S \) by:

\[
\tilde{S} = \{(w, \theta) | H \cdot (w, \theta) \leq C\}
\]

A key feature of this algorithm is that we discretize the action space, i.e., we create a grid of possible values for \( m \) and \( h \) (note that \( x \) is implied by \( m \) and \( h \)). This discretization simplifies computation of \( \tilde{S} \) by allowing us to find it by solving a sequence of linear programs.

The outer hyperplane approximation algorithm proceeds as follows:

1. Initialize subgradients, \( H \), and hyperplane levels, \( C_0 \).
2. Given a set of subgradients, \( H \), and hyperplane levels, \( C_t \), for each subgradient \( h_i \in H \):
   - Solve a linear program (described below) for each action in the action space.
   - Find the maximum and update the corresponding hyperplane level, \( C_{i,t+1} \).
3. If \( |C_{t+1} - C_t| > \epsilon \), return to 2.

**Step 1** simply creates a large initial set \( S_0 \).

Given some set \( S_t \), **Step 2** then constructs the set \( S_{t+1} = D(S_t) \). The linear program in Step 2 is designed to construct a set \( S_{t+1} \) that is as large as possible while satisfying the constraints of the \( D(S) \) operator.

To do this, for each subgradient \( h_i \), and for each point in the action space \((m_j, h_j)\), we solve the following problem:

\[
\max_{w', \theta'} h_i \cdot (w, \theta)
\]

subject to

\[
H \cdot (w', \theta') \leq C_t \\
w = u(f(x_j)) + v(m_j) + \beta w'
\]
\[
\theta = u'(f(x_j))(m_j + x_j) \\
x_j = m_j(h_j - 1) \\
m_j(u'(f(x_j)) - v'(m_j)) \leq \beta \theta' \quad (= \text{if } m_j < \bar{m})
\]

This problem maximizes the hyperplane level for a given set of actions.

The second part of Step 2 then finds the maximum possible hyperplane level across the action space.

The algorithm constructs a sequence of progressively smaller sets \(S_{t+1} \subset S_t \subset S_{t-1} \cdots \subset S_0\)

**Step 3** ends the algorithm when the difference between these sets is small enough.

We have created a Python class that solves the model assuming the following functional forms:

\[
u(c) = \log(c) \\
v(m) = \frac{1}{500} (m\bar{m} - 0.5m^2)^{0.5} \\
f(x) = 180 - (0.4x)^2
\]

The remaining parameters \(\{\beta, \bar{m}, h, h\}\) 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)

Note: this code requires the *polytope* package.

The package can be installed in a terminal/command prompt with pip

```
pip install polytope
```

```python
""
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.

```python
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)
    f = lambda 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):
```

9.8. Competitive Equilibria of Chang Model
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 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):
    ""
    ""
    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):
    ""
    ""

    # 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,
                              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,
->aeq_S,
->self.p_bnds_s)
if res.status == 0:
->beta * res.x[0]) + self.H[i, 1] * self.Theta_vec[j]
    t_alaa2_s[j] = res.x
idx_c = np.where(c_alaa2_c == max(c_alaa2_c))[0][0]
self.z1_c[:, i] = np.array([self.u_vec[idx_c] + self.beta * t_alaa2_s,
->c[idx_c, 0],
                       self.Theta_vec[idx_c]])
idx_s = np.where(c_alaa2_s == max(c_alaa2_s))[0][0]
self.z1_s[:, i] = np.array([self.u_vec[idx_s] + self.beta * t_alaa2_s,
->s[idx_s, 0],
                       self.Theta_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('### --------------- ###
    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 theta
        wmin_c, wmax_c = np.min(self.z1_c, axis=1)[0], np.max(self.z1_c, axis=1)[0]

9.8. Competitive Equilibria of Chang Model 1289
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

effapsed = 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 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] + 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[0] - 1) + v_p(x[1]) * x[1] + self.β * x[2] - θ}
          * x[1] - θ))
cons2 = ({'type': 'ineq', 'fun': lambda x: uc_p(f(x[0], mbar)) * mbar + v_p(mbar) * mbar + self.β * x[1] - θ,
           'type': 'eq', 'fun': lambda x: uc_p(f(x[0], mbar)) * x[0] + (x[0] - 1) + v_p(mbar) * mbar + self.β * x[1] - θ}
          * 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
cl = np.linalg.solve(Φ, p_iter1)

# 3. Compute distance and update
diff = np.linalg.norm(c - cl)
if bool(disp == True):
    print(diff)
c = np.copy(cl)
    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)

c1 = ChangModel(β=0.3, mbar=30, h_min=0.9, h_max=2, n_h=8, n_m=35, N_g=10)
c1.solve_sustainable()

### --------------- ###
Solving Chang Model using Outer Hyperplane Approximation
### --------------- ###

Maximum difference when updating hyperplane levels:
[ 1.9168]
[ 0.66782]
[ 0.49235]
[ 0.32412]
[ 0.19022]
[ 0.10863]
[ 0.05817]
[ 0.0262]
[ 0.01836]
[ 0.01415]
[ 0.00297]
[ 0.00089]
[ 0.00027]
[ 0.00008]
[ 0.00002]
[ 0.00001]
Convergence achieved after 16 iterations and 349.53 second

import polytope
import matplotlib.pyplot as plt

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)

ch2 = ChangModel(β=0.8, mbar=30, h_min=0.9, h_max=1/0.8, n_h=8, n_m=35, N_g=10)
ch2.solve_sustainable()
Solving Chang Model using Outer Hyperplane Approximation

Maximum difference when updating hyperplane levels:
```
[ 0.06369]
[ 0.02476]
[ 0.02153]
[ 0.01915]
[ 0.01795]
[ 0.01642]
[ 0.01507]
[ 0.01284]
[ 0.01106]
[ 0.00694]
[ 0.0085]
[ 0.00781]
[ 0.00433]
[ 0.00492]
[ 0.00303]
[ 0.00182]
[ 0.00638]
[ 0.00116]
[ 0.00093]
[ 0.00075]
[ 0.0006]
[ 0.00494]
[ 0.00038]
[ 0.00121]
[ 0.00024]
[ 0.0002]
[ 0.0016]
[ 0.00013]
[ 0.0001]
[ 0.00008]
[ 0.00006]
[ 0.00005]
[ 0.00004]
[ 0.00003]
[ 0.00003]
[ 0.00002]
[ 0.00002]
[ 0.00001]
[ 0.00001]
[ 0.00001]
```

Convergence achieved after 40 iterations and 849.36 seconds

```
plot_competitive(ch2)
```

Chapter 9. Dynamic Programming Squared
### 9.8.7 Solving a continuation Ramsey planners 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:

$$\theta \leq u'(f(x))x + v'(m)m + \beta \theta'$$

$$\theta = u'(f(x))(m + x)$$

$$x = m(h - 1)$$

$$(m, x, h) \in E$$
To solve this Bellman equation, we must know the set $\Omega$.

We have solved the Bellman equation for the two sets of parameter values for which we computed the equilibrium value sets above.

Hence for these parameter configurations, we know the bounds of $\Omega$.

The two sets of parameters differ only in the level of $\beta$.

From the figures earlier in this lecture, we know that when $\beta = 0.3$, $\Omega = [0.0088, 0.0499]$, and when $\beta = 0.8$, $\Omega = [0.0395, 0.2193]$.

```python
ch1 = ChangModel(\beta=0.3, mbar=30, h_min=0.99, h_max=1/0.3, n_h=8, n_m=35, N_g=50)
ch2 = ChangModel(\beta=0.8, mbar=30, h_min=0.1, h_max=1/0.8, n_h=20, n_m=50, N_g=50)
```

```python
ch1.solve_bellman(\theta_min=0.01, \theta_max=0.0499, order=30, tol=1e-6)
ch2.solve_bellman(\theta_min=0.045, \theta_max=0.15, order=30, tol=1e-6)
```

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:

```python
max(abs(ch1.resid_grid)), max(abs(ch2.resid_grid))
```

(6.463130983291876e-06, 7.0466161972149166e-07)

The value functions plotted below trace out the right edges of the sets of equilibrium values plotted above.

```python
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
for ax, model in zip(axes, (ch1, ch2)):
    ax.plot(model.\theta_grid, model.p_iter)
    ax.set(xlabel='$\theta$',
           ylabel='$J(\theta)$',
           title=rf'$\beta = {model.\beta}$')
plt.show()
```
The next figure plots the optimal policy functions; values of $\theta'$, $m$, $x$, $h$ for each value of the state $\theta$:

```python
for model in (ch1, ch2):
    fig, axes = plt.subplots(2, 2, figsize=(12, 6), sharex=True)
    fig.suptitle(rf"$\beta = {model.\beta}$", fontsize=16)
    plots = [model.\theta_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.\theta_grid_fine, plot)
        ax.set_xlabel(r"\theta", fontsize=14)
        ax.set_ylabel(label, fontsize=14)
    plt.show()
```

9.8. Competitive Equilibria of Chang Model
With the first set of parameter values, the value of \( \theta' \) chosen by the Ramsey planner quickly hits the upper limit of \( \Omega \).

But with the second set of parameters it converges to a value in the interior of the set.

Consequently, the choice of \( \tilde{\theta} \) 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 $\tilde{\theta}$, whereas in the second set of parameter values, it intersects in the interior

```python
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

for ax, model in zip(axes, (ch1, ch2)):
    ax.plot(model._grid_fine, model._prime_grid, label=r'$\theta'(\theta)$')
    ax.plot(model._grid_fine, model._grid_fine, label=r'$\theta$')
    ax.set(xlabel=r'$\theta$', title=rf'$\beta = {model.\beta}$')

axes[0].legend()
plt.show()
```

Subproblem 2 is equivalent to the planner choosing the initial value of $\theta$ (i.e. the value which maximizes the value function)

From this starting point, we can then trace out the paths for $\{\theta_t, m_t, h_t, x_t\}_{t=0}^\infty$ that support this equilibrium

These are shown below for both sets of parameters

```python
for model in (ch1, ch2):
    fig, axes = plt.subplots(2, 2, figsize=(12, 6))
    fig.suptitle(rf'$\beta = {model.\beta}$')
    plots = [model.\theta_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()
```

9.8. Competitive Equilibria of Chang Model
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

### 9.9 Credible Government Policies in Chang Model

#### Contents

- Credible Government Policies in Chang Model
  - Overview
  - The setting
  - Calculating set of sustainable \((w, \theta)\) pairs

Co-author: Sebastian Graves

#### 9.9.1 Overview

Some of the material in this lecture and competitive equilibria in the Chang model can be viewed as more sophisticated and complete treatments of the topics discussed in Ramsey plans, time inconsistency, sustainable plans.

This lecture assumes almost the same economic environment analyzed in competitive equilibria in the Chang model.

The only change – and it is a substantial one – is the timing protocol for making government decisions.

In competitive equilibria in the Chang model, a Ramsey planner chose a comprehensive government policy once-and-for-all at time 0.

Now in this lecture, there is no time 0 Ramsey planner.

Instead there is a sequence of government decision makers, one for each \( t \).

The time \( t \) government decision maker choose time \( t \) government actions after forecasting what future governments will do.

We use the notion of a sustainable plan proposed in [CK90], also referred to as a credible public policy in [Sto89].

Technically, this lecture starts where lecture competitive equilibria in the Chang model on Ramsey plans within the Chang [Cha98] model stopped.

That lecture presents recursive representations of competitive equilibria and a Ramsey plan for a version of a model of Calvo [Cal78] that Chang used to analyze and illustrate these concepts.

We used two operators to characterize competitive equilibria and a Ramsey plan, respectively.

In this lecture, we define a credible public policy or sustainable plan.

Starting from a large enough initial set \( Z_0 \), we use iterations on Changs set-to-set operator \( \tilde{D}(Z) \) to compute a set of values associated with sustainable plans.
Chang's operator $\hat{D}(Z)$ is closely connected with the operator $D(Z)$ introduced in lecture competitive equilibria in the Chang model

- $\hat{D}(Z)$ 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

9.9.2 The setting

We begin by reviewing the set up deployed in competitive equilibria in the Chang model

Chang's model, adopted from Calvo, is designed to focus on the intertemporal trade-offs between the welfare benefits of deflation and the welfare costs associated with the high tax collections required to retire money at a rate that delivers deflation

A benevolent time 0 government can promote utility generating increases in real balances only by imposing an infinite sequence of sufficiently large distorting tax collections

To promote the welfare increasing effects of high real balances, the government wants to induce gradual deflation

We start by reviewing notation

For a sequence of scalars $\vec{z} = \{z_t\}_{t=0}^{\infty}$, let $\vec{z}^t = (z_0, \ldots, z_t)$, $\vec{z}_t = (z_t, z_{t+1}, \ldots)$.

An infinitely lived representative agent and an infinitely lived government exist at dates $t = 0, 1, \ldots$

The objects in play are

- an initial quantity $M_{-1}$ of nominal money holdings
- a sequence of inverse money growth rates $\vec{h}$ and an associated sequence of nominal money holdings $\vec{M}$
- a sequence of values of money $\vec{q}$
- a sequence of real money holdings $\vec{m}$
- a sequence of total tax collections $\vec{x}$
- a sequence of per capita rates of consumption $\vec{c}$
- a sequence of per capita incomes $\vec{y}$

A benevolent government chooses sequences $(\vec{M}, \vec{h}, \vec{x})$ subject to a sequence of budget constraints and other constraints imposed by competitive equilibrium

Given tax collection and price of money sequences, a representative household chooses sequences $(\vec{c}, \vec{m})$ of consumption and real balances

In competitive equilibrium, the price of money sequence $\vec{q}$ clears markets, thereby reconciling decisions of the government and the representative household
The household 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

$$\sum_{t=0}^{\infty} \beta^t [u(c_t) + v(q_t M_t)]$$

subject to

$$q_t M_t \leq y_t + q_t M_{t-1} - c_t - x_t$$

and

$$q_t M_t \leq \bar{m}$$

Here $q_t$ is the reciprocal of the price level at $t$, also known as the value of money.

Chang [Cha98] assumes that

- $u : \mathbb{R}_+ \to \mathbb{R}$ is twice continuously differentiable, strictly concave, and strictly increasing;
- $v : \mathbb{R}_+ \to \mathbb{R}$ is twice continuously differentiable and strictly concave;
- $u'(c)_{c \to 0} = \lim_{m \to 0} v'(m) = +\infty$;
- there is a finite level $m = m^f$ such that $v'(m^f) = 0$.

Real balances carried out of a period equal $m_t = q_t M_t$.

Inequality (9.173) is the households 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 (9.174) imposes an exogenous upper bound $\bar{m}$ on the choice of real balances, where $\bar{m} \geq m^f$.

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

\[
-x_t = m_t(1 - h_t) \tag{9.175}
\]

The restrictions \( m_t \in [0, \bar{m}] \) and \( h_t \in \Pi \) evidently imply that \( x_t \in X \equiv [(\pi - 1)\bar{m}, (\pi - 1)\bar{m}] \)

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:

\[
y_t = f(x_t) \tag{9.176}
\]

where \( f: \mathbb{R} \rightarrow \mathbb{R} \) 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 (9.172)

**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 \( \bar{q} \) and its expectations about future values of \( x \) and \( y \), the household chooses \( M_t \) and therefore \( m_t = q_tM_t \);
- 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

**Households problem**

Given \( M_{t-1} \) and \( \{q_t\}_{t=0}^\infty \), the households problem is

\[
\mathcal{L} = \max_{c_t, \tilde{M}} \min_{\lambda, \bar{\mu}} \sum_{t=0}^\infty \beta^t \left \{ u(c_t) + v(M_t q_t) + \lambda_t[y_t - c_t - x_t + q_t M_{t-1} - q_t M_t] + \mu_t (\bar{m} - q_t \bar{M}_t) \right \}
\]
First-order conditions with respect to $c_t$ and $M_t$, respectively, are

\[
\begin{align*}
u'(c_t) &= \lambda_t \\
q_t[u'(c_t) - v'(M_t q_t)] &\leq \beta u'(c_{t+1}) q_{t+1}, \quad \text{if } M_t q_t < \bar{m}
\end{align*}
\]

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

\[
m_t[u'(c_t) - v'(m_t)] \leq \beta u'(f(x_{t+1})) m_{t+1} h_{t+1}, \quad \text{if } m_t < \bar{m}
\]

(9.177)

Define the following key variable

\[
\theta_{t+1} = u'(f(x_{t+1})) m_{t+1} h_{t+1}
\]

(9.178)

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 (9.178) shows that \( \theta_{t+1} \) intermediates the influences of \((\vec{x}_{t+1}, \vec{m}_{t+1})\) on the households choice of real balances \( m_t \).

By intermediates we mean that the future paths \((\vec{x}_{t+1}, \vec{m}_{t+1})\) influence \( m_t \) entirely through their effects on the scalar \( \theta_{t+1} \).

The observation that the one dimensional promised marginal utility of real balances \( \theta_{t+1} \) functions in this way is an important step in constructing a class of competitive equilibria that have a recursive representation.

A closely related observation pervaded the analysis of Stackelberg plans in dynamic Stackelberg problems and the Calvo model.

**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 households problem
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 \( w = \sum_{t=0}^{\infty} \beta^t [u(c_t) + v(q_t, M_t)] \) be a value associated with a particular competitive equilibrium

• A recursive representation of a credible government policy is a pair of initial conditions \((w_0, \theta_0)\) and a five-tuple of functions

\[
\begin{align*}
h(w_t, \theta_t), m(h_t, w_t, \theta_t), x(h_t, w_t, \theta_t), \chi(h_t, w_t, \theta_t), \Psi(h_t, w_t, \theta_t)
\end{align*}
\]

mapping \( w_t, \theta_t \) and in some cases \( h_t \) into \( \hat{h}_t, m_t, x_t, w_{t+1}, \) and \( \theta_{t+1} \), respectively

• Starting from initial condition \((w_0, \theta_0)\), a credible government policy can be constructed by iterating on these functions in the following order that respects the within-period timing:

\[
\begin{align*}
\hat{h}_t &= h(w_t, \theta_t) \\
m_t &= m(h_t, w_t, \theta_t) \\
x_t &= x(h_t, w_t, \theta_t) \\
w_{t+1} &= \chi(h_t, w_t, \theta_t) \\
\theta_{t+1} &= \Psi(h_t, w_t, \theta_t)
\end{align*}
\]

(9.179)

• 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 \( t \) governments 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,

\[
\begin{align*}
u(f(x(\hat{h}_t, w_t, \theta_t))) &+ v(m(\hat{h}_t, w_t, \theta_t)) + \beta \chi(\hat{h}_t, w_t, \theta_t) \\
&\geq u(f(x(h_t, w_t, \theta_t))) + v(m(h_t, w_t, \theta_t)) + \beta \chi(h_t, w_t, \theta_t)
\end{align*}
\]

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 sectors expectation about its time \( t \) choice \( \hat{h}_t \)
A credible plan induces the government to confirm the private sector's expectation. The recursive representation of the plan uses the evolution of continuation values to deter the government from wanting to disappoint the private sector's expectations. Technically, a Ramsey plan and a credible plan both incorporate history dependence. For a Ramsey plan, this is encoded in the dynamics of the state variable $\theta_t$, a promised marginal utility that the Ramsey plan delivers to the private sector. For a credible government plan, we the two-dimensional state vector $(w_t, \theta_t)$ encodes history dependence.

### Sustainable plans

A government strategy $\sigma$ and an allocation rule $\alpha$ are said to constitute a sustainable plan (SP) if:

1. $\sigma$ is admissible
2. Given $\sigma$, $\alpha$ is competitive
3. After any history $\tilde{h}^{t-1}$, the continuation of $\sigma$ is optimal for the government; i.e., the sequence $\tilde{h}_t$ induced by $\sigma$ after $\tilde{h}^{t-1}$ maximizes over $CE_\sigma$ given $\alpha$

Given any history $\tilde{h}^{t-1}$, the continuation of a sustainable plan is a sustainable plan. Let $\Theta = \{(\tilde{m}, \tilde{x}, \tilde{h}) \in CE : \text{there is an SP whose outcome is } (\tilde{m}, \tilde{x}, \tilde{h})\}$

Sustainable outcomes are elements of $\Theta$

Now consider the space

$$S = \{(w, \theta) : \text{there is a sustainable outcome } (\tilde{m}, \tilde{x}, \tilde{h}) \in \Theta\}$$

with value

$$w = \sum_{t=0}^{\infty} \beta^t [u(f(x_t)) + v(m_t)] \text{ and such that } u'(f(x_0))(m_0 + x_0) = \theta$$

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 $(h, x) \rightarrow m = qM \rightarrow y = c$.

With this timing protocol in mind, the time 0 component of an SP has the following components:

1. A period 0 action $\tilde{h} \in \Pi$ that the public expects the government to take, together with subsequent within-period consequences $m(\tilde{h}), x(\tilde{h})$ when the government acts as expected
2. For any first period action $h \neq \tilde{h}$ with $h \in CE^0_\pi$, a pair of within-period consequences $m(h), x(h)$ when the government does not act as the public had expected
3. For every $h \in \Pi$, a pair $(w'(h), \theta'(h)) \in S$ to carry into next period
These components must be such that it is optimal for the government to choose \( \hat{h} \) as expected; and for every possible \( h \in \Pi \), the government budget constraint and the households Euler equation must hold with continuation \( \theta \) being \( \theta'(h) \)

Given the timing protocol within the model, the representative households response to a government deviation to \( h \neq \hat{h} \) from a prescribed \( \hat{h} \) consists of a first period action \( m(h) \) and associated subsequent actions, together with future equilibrium prices, captured by \( (w'(h), \theta'(h)) \)

At this point, Chang introduces an idea in the spirit of Abreu, Pearce, and Stacchetti \([APS90]\)

Let \( Z \) be a nonempty subset of \( W \times \Omega \)

Think of using pairs \((w', \theta')\) drawn from \( Z \) as candidate continuation value, promised marginal utility pairs

Define the following operator:

\[
\tilde{D}(Z) = \left\{ (w, \theta) : \text{there is } \hat{h} \in CE^0_\pi \text{ and for each } h \in CE^0_\pi \text{ a four-tuple } (m(h), x(h), w'(h), \theta'(h)) \in [0, \bar{m}] \times X \times Z \right\} 
\]

such that

\[ w = u(f(x(h))) + v(m(\hat{h})) + \beta w'(\hat{h}) \quad (9.181) \]

\[ \theta = u'(f(x(\hat{h}))(m(\hat{h}) + x(\hat{h})) \quad (9.182) \]

and for all \( h \in CE^0_\pi \)

\[ w \geq u(f(x(h))) + v(m(h)) + \beta w'(h) \quad (9.183) \]

\[ x(h) = m(h)(h - 1) \quad (9.184) \]

and

\[ m(h)(u'(f(x(h)) - v'(m(h)))) \leq \beta \theta'(h) \quad (9.185) \]

with equality if \( m(h) < \bar{m} \}

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 \((9.183)\) 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 (9.179), which we repeat here for convenience:

\[
\begin{align*}
\hat{h}_t &= h(w_t, \theta_t) \\
m_t &= m(h_t, w_t, \theta_t) \\
x_t &= x(h_t, w_t, \theta_t) \\
w_{t+1} &= \chi(h_t, w_t, \theta_t) \\
\theta_{t+1} &= \Psi(h_t, w_t, \theta_t)
\end{align*}
\]

**9.9.3 Calculating set of sustainable \((w, \theta)\) pairs**

Above we defined the \( \tilde{D}(Z) \) operator as (9.180)

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 sectors 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

\[
x = m(h - 1)
\]

\[
m(h)(u'(f(x(h))) + v'(m(h))) \leq \beta \theta'(h) \text{ (with equality if } m(h) < \tilde{m})\}
\]

\[
(m, x, w', \theta') \in [0, \tilde{m}] \times X \times Z
\]

For a given deviation \( h \), this problem finds the worst possible sustainable value

9.9. Credible Government Policies in Chang Model
We then define:

\[ BR(Z) = \max P(h; Z) \text{ subject to } h \in CE_π^0 \]

\( BR(Z) \) 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:

\[ E(Z) = \left\{ (w, \theta) : \exists h \in CE_π^0 \text{ and } (m(h), x(h), w'(h), \theta'(h)) \in [0, \tilde{m}] \times X \times Z \right\} \]

such that
\[
\begin{align*}
  w &= u(f(x(h))) + v(m(h)) + \beta w'(h) \\
  \theta &= u'(f(x(h)))(m(h) + x(h)) \\
  x(h) &= m(h)(h - 1) \\
  m(h)(u'(f(x(h))) - v'(m(h))) &\leq \beta \theta'(h) \text{ (with equality if } m(h) < \tilde{m})
\end{align*}
\]

and
\[ w \geq BR(Z) \}

Aside from the final incentive constraint, this is the same as the operator in competitive equilibria in the Chang model.

Consequently, to implement this operator we just need to add one step to our outer hyperplane approximation algorithm:

1. Initialize subgradients, \( H \), and hyperplane levels, \( C_0 \)
2. Given a set of subgradients, \( H \), and hyperplane levels, \( C_t \), calculate \( BR(S_t) \)
3. Given \( H \), \( C_t \), and \( BR(S_t) \), for each subgradient \( h_i \in H \):
   - Solve a linear program (described below) for each action in the action space
   - Find the maximum and update the corresponding hyperplane level, \( C_{i,t+1} \)
4. If \( |C_{t+1} - C_t| > \epsilon \), return to 2

**Step 1** simply creates a large initial set \( S_0 \).

Given some set \( S_t \), **Step 2** then constructs the value \( BR(S_t) \).

To do this, we solve the following problem for each point in the action space \( (m_j, h_j) \):

\[
\min_{[w', \theta']} u(f(x_j)) + v(m_j) + \beta w'
\]

subject to
\[
\begin{align*}
  H \cdot (w', \theta') &\leq C_t \\
  x_j &= m_j(h_j - 1)
\end{align*}
\]
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:

$$
\begin{align*}
\max_{[w', \theta']} & \quad h_i \cdot (w, \theta) \\
\text{subject to} & \quad H \cdot (w', \theta') \leq C_t \\
& \quad w = u(f(x)) + v(m_j) + \beta w' \\
& \quad \theta = u'(f(x))(m_j + x_j) \\
& \quad x_j = m_j(h_j - 1) \\
& \quad m_j(u'(f(x)) - v'(m_j)) \leq \beta \theta' \quad (= \text{if } m_j < \bar{m}) \\
& \quad w \geq BR(Z)
\end{align*}
$$

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:

$$
\begin{align*}
u(c) &= \log(c) \\
v(m) &= \frac{1}{500} (m \bar{m} - 0.5 m^2)^{0.5} \\
f(x) &= 180 - (0.4x)^2
\end{align*}
$$

The remaining parameters $\{\beta, \bar{m}, h, \bar{h}\}$ 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.

Note: this code requires the polytope package.

The package can be installed in a terminal/command prompt with pip.

---

9.9. Credible Government Policies in Chang Model
pip install polytope

""
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 m: 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])))
self.u_vec = u(self.A[:, 0], self.A[:, 1])
self._vec = (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 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[:, i]
    Z[1, i] = o[1] + r*H[:, 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_s, self.c0_c, 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):
    ""
    ""

    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.beta])))
    bineq_mbar = np.vstack((self.c0_s, 0))
    aineq = self.H
    bineq = self.c0_s
    aeq = [[0, -self.beta]]

    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
```python
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, A_eq=aeq, b_ub=bineq, 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):
    # 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_ala2_c, t_ala2_c = np.full(self.N_a, -np.inf), np.zeros((self.N_a, 2))
        c_ala2_s, t_ala2_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]
```

9.9. Credible Government Policies in Chang Model
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('### ----------------- ###

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 theta(h, m):
    x = m * (h - 1)
    theta = uc_p(f(h, m)) * (m + x)
    return theta

# Bounds for Maximization
lb1 = np.array([self.h_min, 0, self.mbar - 1e-5])
ub1 = np.array([self.h_max, self.mbar, self.h_max])

lb2 = np.array([self.h_min, self.h_min])
ub2 = np.array([self.h_max, self.h_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 = theta_min + (roots - 1) / 2 * (theta_max - theta_min)
# Create basis matrix
Phi = cheb.chebvander(roots, order - 1)
c = np.zeros(Phi.shape[0])

# Function to minimize and constraints
def p_fun(x):
    scale = -1 + 2 * (x[2] - theta_min)/(theta_max - theta_min)
p_fun = - (u(x[0], x[1]) + self.beta * np.dot(cheb.chebvander(scale, order - 1), c))
    return p_fun

def p_fun2(x):
    scale = -1 + 2*(x[1] - theta_min)/(theta_max - theta_min)
p_fun = - (u(x[0], mbar) + self.beta * 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(x[1]) * x[1] + self.beta * x[2] - theta,
        'type': 'eq', 'fun': lambda x: uc_p(f(x[0], x[1])) * x[0] - (x[1] - theta)})

cons2 = ({'type': 'ineq', 'fun': lambda x: uc_p(f(x[0], mbar)) * mbar - (x[0] - 1) + v(mbar) + mbar + self.beta * x[1] - theta,
        'type': 'eq', 'fun': lambda x: uc_p(f(x[0], mbar)) * x[0] - mbar - theta})
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):
    \( \theta = \theta_{\text{grid fine}}[i] \)
    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
        \( \theta_{\text{prime grid}}[i] = \text{res.x}[2] \)
        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
        \( \theta_{\text{prime grid}}[i] = \text{res.x}[1] \)
        h_grid[i] = res.x[0]
        m_grid[i] = self.mbar
        scale = -1 + 2 * (\text{\theta} - \text{\theta}_{\text{min}})/(\text{\theta}_{\text{max}} - \text{\theta}_{\text{min}})
        resid_grid[i] = np.dot(cheb.chebvander(scale, order-1), c) - p

self.resid_grid = resid_grid
self.\theta_{\text{grid fine}} = \theta_{\text{grid fine}}
self.\theta_{\text{prime grid}} = \theta_{\text{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
\theta_{\text{series}} = np.zeros(31)
m_series = np.zeros(30)
h_series = np.zeros(30)

# Find initial \theta
def ValFun(x):
    scale = -1 + 2*(x - \text{\theta}_{\text{min}})/(\text{\theta}_{\text{max}} - \text{\theta}_{\text{min}})
    p_fun = np.dot(cheb.chebvander(scale, order - 1), c)
    return -p_fun

res = minimize(ValFun,
                (\text{\theta}_{\text{min}} + \text{\theta}_{\text{max}})/2,
                bounds=[(\text{\theta}_{\text{min}}, \text{\theta}_{\text{max}})])
\theta_{\text{series}}[0] = \text{res}\cdot \text{x}

# Simulate
for i in range(30):
    \theta = \theta_{\text{series}}[i]
    \text{res} = \text{minimize}(p\_\text{fun},
        lb1 + (ub1-lb1)/2,
        \text{method}='\text{SLSQP}',
        \text{bounds}=\text{bnds1},
        \text{constraints}=\text{cons1},
        \text{tol}=1e-10)
    \text{if res.success == \text{True}:}
        \text{p} = -p\_\text{fun}(\text{res}\cdot \text{x})
        h\_\text{series}\[i] = \text{res}\cdot \text{x}[0]
        m\_\text{series}\[i] = \text{res}\cdot \text{x}[1]
        \theta\_\text{series}[i+1] = \text{res}\cdot \text{x}[2]
    \text{res2} = \text{minimize}(p\_\text{fun2},
        lb2 + (ub2-lb2)/2,
        \text{method}='\text{SLSQP}',
        \text{bounds}=\text{bnds2},
        \text{constraints}=\text{cons2},
        \text{tol}=1e-10)
    \text{if -p\_\text{fun2}(res2.x) > p and res2.success == \text{True:}
        h\_\text{series}\[i] = \text{res2}\cdot \text{x}[0]
        m\_\text{series}\[i] = \text{self}\cdot \text{mbar}
        \theta\_\text{series}[i+1] = \text{res2}\cdot \text{x}[1]

    \text{self}\cdot \theta\_\text{series} = \theta\_\text{series}
    \text{self}\cdot m\_\text{series} = m\_\text{series}
    \text{self}\cdot h\_\text{series} = h\_\text{series}
    \text{self}\cdot x\_\text{series} = m\_\text{series} \cdot (h\_\text{series} - 1)

**Comparison of sets**

The set of \((w, \theta)\) associated with sustainable plans is smaller than the set of \((w, \theta)\) pairs associated with competitive equilibria, since the additional constraints associated with sustainability must also be satisfied

Let's compute two examples, one with a low \(\beta\), another with a higher \(\beta\)

```python
ch1 = ChangModel(\beta=0.3, mbar=30, h_min=0.9, h_max=2, n_h=8, n_m=35, N_g=10)
```

```python
ch1.solve_sustainable()
```

```python
### --------------- ###
Solving Chang Model using Outer Hyperplane Approximation
### --------------- ###
Maximum difference when updating hyperplane levels:
[ 1.9168]
[ 0.66782]
[ 0.49235]
```
Convergence achieved after 16 iterations and 522.52 second

The following plot shows both the set of $w, \theta$ pairs associated with competitive equilibria (in red) and the smaller set of $w, \theta$ pairs associated with sustainable plans (in blue)

```python
import polytope
import matplotlib.pyplot as plt

def plot_equilibria(ChangModel):
    
    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]),
                xtext=(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:

```python
c2 = ChangModel(\( \beta = 0.8 \), mbar=30, h_min=0.9, h_max=1/0.8, n_h=8, n_m=35, N_g=10)
c2.solve_sustainable()
```

### --------------- ###
Solving Chang Model using Outer Hyperplane Approximation
### --------------- ###

Maximum difference when updating hyperplane levels:

```
[ 0.06369]
[ 0.02476]
[ 0.02153]
[ 0.01915]
[ 0.01795]
[ 0.01642]
[ 0.01507]
[ 0.01284]
[ 0.01106]
[ 0.00694]
```
(0.0085
(0.00781
(0.00433
(0.00492
(0.00303
(0.00182
(0.00638
(0.00116
(0.00093
(0.00075
(0.0006
(0.00494
(0.00038
(0.00121
(0.00024
(0.0002
(0.00016
(0.00013
(0.0001
(0.00008
(0.00006
(0.00005
(0.00004
(0.00003
(0.00003
(0.00002
(0.00002
(0.00001
(0.00001
(0.00001
Convergence achieved after 40 iterations and 1258.26 seconds

Let's plot both sets

plot_equilibria(ch2)
Evidently, the Ramsey plan is now sustainable
This section of the course contains foundational mathematical and statistical tools and techniques
This section of the course contains foundational models for dynamic economic modeling. Most are single agent problems that take the activities of other agents as given. Later we will look at full equilibrium problems.
MULTIPLE AGENT MODELS

These lectures look at important economic models that also illustrate common equilibrium concepts.

Acknowledgements: These lectures have benefitted greatly from comments and suggestion from our colleagues, students and friends. Special thanks go to Anmol Bhandari, Long Bui, Jeong-Hun Choi, Chase Coleman, David Evans, Shunsuke Hori, Chenghan Hou, Doc-Jin Jang, Spencer Lyon, Qingyin Ma, Akira Matsushita, Matthew McKay, Tomohito Okabe, Alex Olssen, Nathan Palmer and Yixiao Zhou.
This section of the course contains foundational mathematical and statistical tools and techniques.
This section of the course contains foundational models for dynamic economic modeling. Most are single agent problems that take the activities of other agents as given. Later we will look at full equilibrium problems.
These lectures look at important economic models that also illustrate common equilibrium concepts.
These lectures look at important concepts in time series that are used in economics.

Acknowledgements: These lectures have benefitted greatly from comments and suggestion from our colleagues, students and friends. Special thanks go to Anmol Bhandari, Long Bui, Jeong-Hun Choi, Chase Coleman, David Evans, Shunsuke Hori, Chenghan Hou, Doc-Jin Jang, Spencer Lyon, Qingyin Ma, Akira Matsushita, Matthew McKay, Tomohito Okabe, Alex Olssen, Nathan Palmer and Yixiao Zhou.

Acknowledgements: These lectures have benefitted greatly from comments and suggestion from our colleagues, students and friends. Special thanks go to Anmol Bhandari, Long Bui, Jeong-Hun Choi, Chase Coleman, David Evans, Shunsuke Hori, Chenghan Hou, Doc-Jin Jang, Spencer Lyon, Qingyin Ma, Akira Matsushita, Matthew McKay, Tomohito Okabe, Alex Olssen, Nathan Palmer and Yixiao Zhou.


