QuantEcon.lectures-python3 PDF

Release 2017-Jul-30

Thomas J. Sargent and John Stachurski

Jul 30, 2017

PREFACE

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 Part 1: Introduction to Python **1**
1.1 About Python . . . 1
1.2 Setting up Your Python Environment . . . 14
1.3 An Introductory Example . . . 37
1.4 Python Essentials . . . 55
1.5 NumPy . . . 75
1.6 Matplotlib . . . 95
1.7 SciPy . . . 107

2 Part 2: Advanced Python Programming **119**
2.1 Object Oriented Programming . . . 119
2.2 How it Works: Data, Variables and Names . . . 138
2.3 More Language Features . . . 156
2.4 Debugging . . . 175
2.5 Writing Good Code . . . 180
2.6 The Need for Speed . . . 187

3 Part 3: Data and Empirics **207**
3.1 Pandas . . . 207
3.2 Pandas for Panel Data . . . 223
3.3 Linear Regression in Python . . . 245

4 Part 4: Tools and Techniques **263**
4.1 Linear Algebra . . . 263
4.2 Orthogonal Projections and Their Applications . . . 283
4.3 Finite Markov Chains . . . 300
4.4 LLN and CLT . . . 323
4.5 Linear State Space Models . . . 342
4.6 Continuous State Markov Chains . . . 368
4.7 A First Look at the Kalman Filter . . . 389
4.8 LQ Dynamic Programming Problems . . . 405
4.9 Covariance Stationary Processes . . . 435
4.10 Estimation of Spectra . . . 450
4.11 Additive Functionals . . . 464
4.12 Multiplicative Functionals . . . 482

4.13 Classical Control with Linear Algebra . . . 504
4.14 Classical Filtering With Linear Algebra . . . 527
4.15 Discrete State Dynamic Programming . . . 547

5 Part 5: Single Agent Models 571
5.1 Shortest Paths . . . 571
5.2 The McCall Job Search Model . . . 579
5.3 A Stochastic Optimal Growth Model . . . 592
5.4 Time Iteration with Euler Equations . . . 611
5.5 The Endogenous Grid Method . . . 628
5.6 The Income Fluctuation Problem . . . 637
5.7 A Problem that Stumped Milton Friedman . . . 655
5.8 The Permanent Income Model: I . . . 674
5.9 The Permanent Income Model: II . . . 689
5.10 Search with Offer Distribution Unknown . . . 708
5.11 Robustness . . . 724

6 Part 6: Multiple Agent Models 745
6.1 Schelling's Segregation Model . . . 745
6.2 A Lake Model of Employment and Unemployment . . . 756
6.3 Uncertainty Traps . . . 784
6.4 Rational Expectations Equilibrium . . . 798
6.5 Markov Perfect Equilibrium . . . 812
6.6 Introduction to Asset Pricing . . . 830
6.7 The Lucas Asset Pricing Model . . . 849
6.8 A Harrison-Kreps Model of Asset Prices . . . 862
6.9 The Aiyagari Model . . . 871
6.10 Consumption and Tax Smoothing with Complete and Incomplete Markets . . . 880
6.11 On-the-Job Search . . . 898

7 Part 7: Topics in Economic Dynamics 911
7.1 Modeling Career Choice . . . 911
7.2 Dynamic Stackelberg Problems . . . 925
7.3 History Dependent Public Policies . . . 939
7.4 Optimal Taxation with State-Contingent Debt . . . 959
7.5 Optimal Taxation in an LQ Economy . . . 992
7.6 Optimal Taxation without State-Contingent Debt . . . 1014
7.7 Default Risk and Income Fluctuations . . . 1041
7.8 Globalization and Cycles . . . 1057

Bibliography 1077

Index 1085

CHAPTER

ONE

PART 1: INTRODUCTION TO PYTHON

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

About Python

Contents

- *About Python*
  - *Overview*
  - *What's Python?*
  - *Scientific Programming*
  - *Learn More*

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

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 the relative popularity of Python

How about in the scientific domain?

Here's a comparison of Python vs MATLAB

Of course Python is a general purpose language, so comparing its popularity against a niche language like MATLAB isn't really fair

To restrict coverage to scientific computing, let's compare MATLAB against pandas, a library for data analysis with Python

Once again, this isn't particulary fair, since it depends where users of different languages tend to ask questions

However, it's probably safe to say that

- Python is one of the most popular programming languages worldwide
- Python is a major tool for scientific computing, accounting for a 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 it's highly beneficial because it makes the syntax easy to read and easy to remember

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

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

Scientific Programming

Over the last decade, Python has become one of the core languages of scientific computing

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

- Machine learning and data science
- Astronomy
- Artificial intelligence
- Chemistry
- Computational biology
- Meteorology
- etc., etc.

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

```
import numpy as np                    # Load the library

a = np.linspace(-np.pi, np.pi, 100)   # Create array (even grid from -pi to pi)
b = np.cos(a)                         # Apply cosine to each element of a
c = np.ones(25)                       # An array of 25 ones
np.dot(c, c)                          # Compute inner product
```

```
25.0
```

The SciPy library is built on top of NumPy and provides additional functionality For example, let's calculate $\int_{-2}^{2} \phi(z)dz$ where ϕ is the standard normal density

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

phi = norm()
value, error = quad(phi.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

```
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

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

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

solve polynomials

```
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

```
from sympy import limit, sin, diff

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

```
oo
```

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

```
1
```

```
diff(sin(x), x)
```

```
cos(x)
```

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

Can easily create tables of derivatives, generate LaTeX output, add it to figures, etc., etc.

Statistics

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

Pandas

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

Pandas is fast, efficient, flexible and well designed

Here's a simple example

```
import pandas as pd
import scipy as sp
np.random.seed(1234)

data = sp.randn(5, 2)  # Create 5x2 matrix of random numbers for toy example
dates = pd.date_range('28/12/2010', periods=5)
df = pd.DataFrame(data, columns=('price', 'weight'), index=dates)
print(df)
```

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

```
df.mean()
```

```
price     0.411768
weight   -0.699135
dtype: float64
```

Other Useful Statistics Libraries

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

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

```
"""
Filename: nx_demo.py
Authors: John Stachurski and Thomas J. Sargent
"""
```

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

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)  # Generate random graph
pos = nx.get_node_attributes(G, 'pos')    # Get positions of nodes
# find node nearest the center point (0.5,0.5)
dists = [(x - 0.5)**2 + (y - 0.5)**2 for x, y in list(pos.values())]
ncenter = np.argmin(dists)
# Plot graph, coloring by path length from central node
p = nx.single_source_shortest_path_length(G, ncenter)
plt.figure()
nx.draw_networkx_edges(G, pos, alpha=0.4)
nx.draw_networkx_nodes(G, pos, nodelist=list(p.keys()),
                       node_size=120, alpha=0.5,
                       node_color=list(p.values()), cmap=plt.cm.jet_r)
plt.show()
```

The figure it produces looks as follows

Cloud Computing

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

A nice example is Wakari

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 Amazon's 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

Learn More

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

Setting up Your Python Environment

Contents

- *Setting up Your Python Environment*
  - *Overview*
  - *First Steps*
  - *Jupyter*
  - *Additional Software*
  - *Alternatives*
  - *Exercises*

Overview

This lecture is intended to be the first step on your Python journey

In it 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 Python programs that underpin these lectures

Important Notes

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

Please read on for instructions

First Steps

While there are several such distributions, we highly recommend 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 (using Python 3.6)
- 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

Package Management

The packages in Anaconda contain the various scientific libraries used in day to day scientific programming

Anaconda supplies a great tool called *conda* to keep your packages organized and up to date

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
    - If you don't know what a terminal is
        - For Mac users, see this guide
        - For Windows users, search for the `cmd` application or see this guide
        - Linux users – you already know what a terminal is
2. Type `conda update anaconda`

(If you've already installed Anaconda and it was a little while ago, please make sure you execute this step)

Another useful command is `conda info`, which tells you about your installation

For more information on *conda*

- type *conda help* in a terminal
- read the documentation online

Get a Modern Browser

We'll 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

Once you've done that we can start having fun

Jupyter

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

Jupyter notebooks provide 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 very 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 QuantEcon's 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

In fact these lectures are designed precisely for executing in Jupyter notebooks

Starting the Jupyter Notebook

To start the Jupyter notebook, either

- search for Jupyter in your applications menu, or
- open up a terminal (*cmd* for Windows) and type `jupyter notebook` (our preferred option)

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)

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 *b* adds a new cell below the current one

To switch to

- command mode from edit mode, hit the `Esc` key or `Ctrl-M`
- edit mode from command mode, hit `Enter` or click in a cell

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

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

A Test Program

Let's run a test program

Here's an arbitrary program we can use: http://matplotlib.org/1.4.1/examples/pie_and_polar_charts/polar_bar_demo.html

On that page you'll see the following code

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

N = 20
theta = np.linspace(0.0, 2 * np.pi, N, endpoint=False)
radii = 10 * np.random.rand(N)
width = np.pi / 4 * 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()
```

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

The easiest way to run this code is to copy and paste 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 we'll work with in depth

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

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

We can explore 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

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

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

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

Now we `Shift+Enter` to produce this

Sharing Notebooks

A notebook can easily be saved and shared between users

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

For example, try downloading the notebook we just created by clicking here

Save it somewhere you can navigate to easily

Now you can import it from the dashboard (the first browser page that opens when you start Jupyter notebook) and run the cells or edit as discussed above

You can also share your notebooks using nbviewer

The notebooks you see there are **static** html representations

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

Once downloaded you can open it as a notebook, as discussed just above

Additional Software

There are some other bits and pieces we need before we can proceed with the lectures

QuantEcon

In these lectures we'll make extensive use of code from the QuantEcon organization

On the Python side we'll 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

Installing QuantEcon.py

You can install QuantEcon.py by typing the following into a terminal (terminal on Mac, cmd on Windows, etc.)

```
pip install quantecon
```

More instructions on installing and keeping your code up to date can be found at QuantEcon

Working with Python Files

How does one run a locally saved Python file using the notebook?

Method 1: Copy and Paste

Copy and paste isn't the slickest way to run programs but sometimes it gets the job done

One option is

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
    - e.g., Notepad, TextEdit, TextMate, depending on your OS
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:

```
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`
- `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

Savings 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

Alternatives

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

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

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

Now

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

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

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

Exercise 2

This exercise will familiare you with git and GitHub

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

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

GitHub is a wonderland of collaborative coding projects

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

Git is the underlying software used to manage these projects

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

There are two main flavors of Git

1. the plain vanilla command line Git version
2. the various point-and-click GUI versions
    - See, for example, the GitHub version

As an exercise, try

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

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

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

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

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

An Introductory Example

Contents

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

We're 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 have none—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

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

The Task: Plotting a White Noise Process

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

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

We'll do this several different ways

Version 1

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

```
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

```
import numpy as np

np.sqrt(4)
```

```
2.0
```

We could also just write

```
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 it's easy to learn and maintain

When you want to do something interesting with Python, you almost always need to import addional 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 it's located in

```
anaconda3/lib/python3.5/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's another way to access NumPy's 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 it's harder for readers to know where *sqrt* came from, should they wish to

Alternative Versions

Let's 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

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

ts_length = 100
epsilon_values = []   # Empty list

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

plt.plot(epsilon_values, 'b-')
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 `epsilon_values` that will store the ϵ_t values as we generate them
- The next three lines are the *for* loop, which repeatedly draws a new random number ϵ_t and appends it to the end of the list `epsilon_values`
- The last two lines generate the plot and display it to the user

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

Lists

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

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

For example, try

```
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)`

```
x
```

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

```
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`

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

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

Another useful list method is `pop()`

```
x
```

```
[10, 'foo', False, 2.5]
```

```
x.pop()
```

```
2.5
```

```
x
```

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

The full set of list methods can be found here

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

```
x
```

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

```
x[0]
```

```
10
```

```
x[1]
```

```
'foo'
```

The For Loop

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

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

```
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

```
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 we'll 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 occuring 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 it's 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

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

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

ts_length = 100
epsilon_values = []
i = 0
while i < ts_length:
    e = np.random.randn()
    epsilon_values.append(e)
    i = i + 1
```

```
plt.plot(epsilon_values, 'b-')
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

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

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

data = generate_data(100)
plt.plot(data, 'b-')
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 `epsilon_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 `epsilon_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 the next piece of code

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


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

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

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

Notes

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

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

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

To understand this, consider the following version

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


def generate_data(n, generator_type):
    epsilon_values = []
    for i in range(n):
        e = generator_type()
        epsilon_values.append(e)
    return epsilon_values

data = generate_data(100, np.random.uniform)
plt.plot(data, 'b-')
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

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

```
7
```

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

```
7
```

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

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

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

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

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

Here's another example

```
range(8)
```

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

```
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

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

into

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

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 π using Monte Carlo. Use no imports besides

```
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 * 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

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

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

Running it produces a figure like so

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 α 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'`

Solutions

Exercise 1

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

```
factorial(4)
```

```
24
```

Exercise 2

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

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

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

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

```
import numpy as np

n = 100000

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

area_estimate = count / n

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

```
3.1496
```

Exercise 4

```
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

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

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, 'b-')
plt.show()
```

Exercise 6

```
alphas = [0.0, 0.8, 0.98]
ts_length = 200

for alpha in alphas:
    x_values = []
    current_x = 0
    for i in range(ts_length):
        x_values.append(current_x)
        current_x = alpha * current_x + np.random.randn()
    plt.plot(x_values, label='alpha = ' + str(alpha))
plt.legend()
plt.show()
```

Python Essentials

Contents

- *Python Essentials*
  - *Overview*
  - *Data Types*
  - *Imports*
  - *Input and Output*
  - *Iterating*
  - *Comparisons and Logical Operators*
  - *More Functions*
  - *Coding Style and PEP8*
  - *Exercises*
  - *Solutions*

In this lecture we'll cover features of the language that are essential to reading and writing Python code

Overview

Topics:

- Data types
- Imports
- Basic file I/O
- The Pythonic approach to iteration
- More on user-defined functions
- Comparisons and logic
- Standard Python style

Data Types

So far we've briefly met several common data types, such as strings, integers, floats and lists

Let's learn a bit more about them

Primitive Data Types

A particularly simple data type is Boolean values, which can be either `True` or `False`

```
x = True
y = 100 < 10    # Python evaluates expression on right and assigns it to y
y
```

```
False
```

```
type(y)
```

```
bool
```

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

```
x + y
```

```
1
```

```
x * y
```

```
0
```

```
True + True
```

```
2
```

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

sum(bools)
```

```
3
```

This is called *Boolean arithmetic* and is very useful in programming

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, interal arithmetic operations on integers are more straightforward

> **Warning:** Be careful: If you're still using Python 2.x, division of two integers **returns only the integer part**

To clarify:

```
1 / 2        # Integer division in Python 2.x
```

```
0            # Returns integer component only
```

For Python 3.x

```
1 / 2        # Division of two integers in Python 3+
```

```
0.5          # Returns a floating point number
```

```
1 // 2        # Integer division in Python 3+
```

```
0
```

Otherwise the following syntax is the same regardless of python version

```
1.0 / 2.0   # Floating point division
```

```
0.5
```

```
1.0 / 2     # Floating point division
```

```
0.5
```

Complex numbers are another primitive data type in Python

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

```
5j
```

There are several more primitive data types that we'll introduce as necessary

Containers

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

We have already discussed lists

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

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

```
('a', 'b')
```

```
type(x)
```

```
tuple
```

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

Lists are mutable while tuples are not

```
x = [1, 2]  # Lists are mutable
x[0] = 10   # Now x = [10, 2], so the list has "mutated"
x = (1, 2)  # Tuples are immutable
x[0] = 10   # Trying to mutate them produces an error
```

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

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

We'll say more about mutable vs immutable a bit later, and explain why the distinction is important

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

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

```
10
```

```
y
```

```
20
```

You've actually *seen an example of this* already

Tuple unpacking is convenient and we'll use it often

Slice Notation

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

For example,

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

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

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

```
[4, 6]
```

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

Negative numbers are also permissible

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

```
[6, 8]
```

The same slice notation works on tuples and strings

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

```
'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

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

```
dict
```

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

```
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

```
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

```
s3 = set(('foo', 'bar', 'foo'))
s3
```

```
set(['foo', 'bar'])  # Unique elements only
```

Imports

From the start, Python has been designed around the twin principles of

- a small core language
- extra functionality in separate *libraries* or *modules*

For example, if you want to compute the square root of an arbitrary number, there's no built in function that will perform this for you

Instead, you need to *import* the functionality from a *module* — in this case a natural choice is `math`

```
import math

math.sqrt(4)
```

```
2.0
```

We discussed the mechanics of importing *earlier*

Note that the `math` module is part of the standard library, which is part of every Python distribution

On the other hand, the scientific libraries we'll work with later are not part of the standard library

We'll talk more about modules as we go along

To end this discussion with a final comment about modules and imports, in your Python travels you will often see the following syntax

```
from math import *

sqrt(4)
```

```
2.0
```

Here `from math import *` pulls all of the functionality of `math` into the current "namespace" — a concept we'll define formally *later on*

Actually this kind of syntax should be avoided for the most part

In essence the reason is that it pulls in lots of variable names without explicitly listing them — a potential source of conflicts

Input and Output

Let's have a quick look at basic file input and output

We discuss only reading and writing to text files

Input and Output

Let's start with writing

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

Here

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

Where is this file that we've created?

Recall that Python maintains a concept of the present working directory (pwd) that can be located by

```
import os
print(os.getcwd())
```

or, in IPython or a Jupyter notebook,

```
%pwd
```

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

You can confirm that the file `newfile.txt` is in your present working directory using a file browser or some other method

(Use `%ls` to list the files in the present working directory)

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

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

```
'Testing\nTesting again'
```

```
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 either specify the full path to the file

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

or change the present working directory to the location of the file via `os.chdir('path_to_file')`

(In python, use `cd` to change directories)

Details are OS specific – a Google search on paths and Python should yield plenty of examples

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, consider the file us_cities.txt, which lists US cities and their population

```
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

The program us_cities.py program reads the data in and makes the conversion:

```
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 = '{0:,}'.format(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

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

```
San Diego      1,326,179
Dallas         1,223,229
```

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 we'll 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,

```
for x in x_values:
    print(x * x)
```

is preferred to

```
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

```
countries = ('Japan', 'Korea', 'China')
cities = ('Tokyo', 'Seoul', 'Beijing')
for country, city in zip(countries, cities):
    print('The capital of {0} is {1}'.format(country, city))
```

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

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

```
{'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

```
letter_list = ['a', 'b', 'c']
for index, letter in enumerate(letter_list):
    print("letter_list[{0}] = '{1}'".format(index, letter))
```

The output of the loop is

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

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

```
1 < 2 < 3
```

```
True
```

```
1 <= 2 <= 3
```

```
True
```

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

```
x = 1     # Assignment
x == 2    # Comparison
```

```
False
```

For "not equal" use !=

```
1 != 2
```

```
True
```

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

```
x = 'yes' if 42 else 'no'
x
```

```
'yes'
```

```
x = 'yes' if [] else '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 and 'f' in 'foo'
```

```
True
```

```
1 < 2 and 'g' in 'foo'
```

```
False
```

```
1 < 2 or 'g' in 'foo'
```

```
True
```

```
not True
```

```
False
```

```
not not True
```

```
True
```

Remember

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

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

```
max(19, 20)
```

```
20
```

```
range(4)
```

```
[0, 1, 2, 3]
```

```
str(22)
```

```
'22'
```

```
type(22)
```

```
int
```

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

```
bools = False, True, True
all(bools)  # True if all are True and False otherwise
```

```
False
```

```
any(bools)  # False if all are False and True otherwise
```

```
True
```

The full list of Python built-ins is here

Now let's talk some more about user-defined functions constructed using the keyword `def`

Why Write Functions?

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

- separating different strands of logic
- facilitating code reuse

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

The basics of user defined functions were discussed *here*

The Flexibility of Python Functions

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

In particular

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

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

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

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

```
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

For example, let's say that this code resides in file `temp.py`

```
# Filename: temp.py
def f(x):
    """
    This function squares its argument
    """
    return x**2
```

After running this code, the docstring is available as follows

```
f?
```

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

```
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

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

and

```
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

```
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

```
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

```
def f(x, coefficients=(1, 1)):
    a, b = coefficients
    return a + b * x
```

After running this code we can call it as follows

```
f(2, coefficients=(0, 0))
```

```
0
```

```
f(2)  # Use default values (1, 1)
```

```
3
```

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

Coding Style and PEP8

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

Among other things, Python strongly favors consistency in programming style

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

In programming, as in mathematics, the opposite is true

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

In Python, the standard style is set out in PEP8

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

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

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n = \sum_{i=0}^{n} a_i x^i \tag{1.1}$$

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] < ... < point[n-1] = b`

Aim for clarity, not efficiency

Solutions

Exercise 1

Part 1 solution:

Here's one possible solution

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

```
6
```

This also works

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

```
6
```

Part 2 solution:

One solution is

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

```
50
```

This also works:

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

```
50
```

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

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

```
50
```

and

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

```
50
```

Part 3 solution

Here's one possibility

```
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

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

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

```
6
```

Exercise 3

Here's one solution:

```
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's a solution:

```
def f(seq_a, seq_b):
    is_subset = True
    for a in seq_a:
        if a not in seq_b:
            is_subset = False
    return is_subset

# == test == #

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

```
True
False
```

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

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

Exercise 5

```
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)
```

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

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

Important Notes

We do of course assume that NumPy is installed on your machine

Moreover, we'll be using the new syntax `A @ B` for matrix multiplication, as opposed to the old syntax `np.dot(A, B)`

This will work if you've installed version Python 3.5 or later of Anaconda

See *this page* for instructions on installing or updating your Anaconda distribution

References

- The official NumPy documentation

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's no need to switch back to C or Fortran

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

```
import numpy as np

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

```
0.49990566939719772
```

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'll discuss code that isn't easy to vectorize and how such routines can also be optimized

NumPy Arrays

The most important thing that NumPy defines is an array data type formally called a 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

```
import numpy as np

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

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

```
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`

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

```
numpy.float64
```

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

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

```
numpy.int64
```

Shape and Dimension

Consider the following assignment

```
z = np.zeros(10)
```

Here `z` is a *flat* array with no dimension — neither row nor column vector

The dimension is recorded in the `shape` attribute, which is a tuple

```
z.shape
```

```
(10,)  # Note syntax for tuple with one element
```

Here the shape tuple has only one element, which is the length of the array (tuples with one element end with a comma)

To give it dimension, we can change the `shape` attribute

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

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

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

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

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

Creating Arrays

As we've seen, the `np.zeros` function creates an array of zeros

You can probably guess what `np.ones` creates

Related is `np.empty`, which creates arrays in memory that can later be populated with data

```
z = np.empty(3)
z
```

```
array([  8.90030222e-307,   4.94944794e+173,   4.04144187e-262])
```

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 = np.linspace(2, 4, 5)  # From 2 to 4, with 5 elements
```

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

```
z = np.identity(2)
z
```

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

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

```
z = np.array([10, 20])                # ndarray from Python list
z
```

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

```
type(z)
```

```
numpy.ndarray
```

```
z = np.array((10, 20), dtype=float)     # Here 'float' is equivalent to 'np.
↪float64'
z
```

```
array([ 10.,  20.])
```

```
z = np.array([[1, 2], [3, 4]])          # 2D array from a list of lists
z
```

```
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

```
na = np.linspace(10, 20, 2)
na is np.asarray(na)   # Does not copy NumPy arrays
```

```
True
```

```
na is np.array(na)     # Does make a new copy --- perhaps unnecessarily
```

```
False
```

To read in the array data from a text file containing numeric data use `np.loadtxt` or `np.genfromtxt`—see the documentation for details

Array Indexing

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

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

z
```

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

```
z[0]
```

```
1.0
```

```
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:

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

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

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

```
1
```

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

```
2
```

And so on

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

Columns and rows can be extracted as follows

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

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

```
z[:,1]
```

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

NumPy arrays of integers can also be used to extract elements

```
z = np.linspace(2, 4, 5)
z
```

```
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
```

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

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

```
array([False,  True,  True, False, False], dtype=bool)
```

```
z[d]
```

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

We'll see why this is useful below

An aside: all elements of an array can be set equal to one number using slice notation

```
z = np.empty(3)
z
```

```
array([ -1.25236750e-041,   0.00000000e+000,   5.45693855e-313])
```

```
z[:] = 42
z
```

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

Array Methods

Arrays have useful methods, all of which are carefully optimized

```
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])
```

```
A.cumprod()            # Cumulative product of the elements of A
```

```
array([ 1,  2,  6, 24])
```

```
A.var()                # Variance
```

```
1.25
```

```
A.std()                # Standard deviation
```

```
1.1180339887498949
```

```
A.shape = (2, 2)
A.T                    # Equivalent to A.transpose()
```

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

Another method worth knowing is `searchsorted()`

If `z` is a nondecreasing array, then `z.searchsorted(a)` returns the index of the first element of `z` that is `>= a`

```
z = np.linspace(2, 4, 5)
z
```

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

```
z.searchsorted(2.2)
```

```
1
```

```
z.searchsorted(2.5)
```

```
1
```

```
z.searchsorted(2.6)
```

```
2
```

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

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

```
np.sum(a)
```

```
10
```

```
np.mean(a)
```

```
2.5
```

Operations on Arrays

Algebraic Operations

The algebraic operators $+$, $-$, $\star$, $/$ and $\star\star$ all act *element-wise* on arrays

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

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

```
a * b
```

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

We can add a scalar to each element as follows

```
a + 10
```

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

Scalar multiplication is similar

```
a * 10
```

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

The two dimensional arrays follow the same general rules

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

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

```
A + 10
```

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

```
A * B
```

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

In particular, `A * 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:

```
import numpy as np

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

```
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

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

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

```
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:

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

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

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

```
array([42,  0])
```

Mutability leads to the following behavior (which causes a surprising amount of outrage in classes that we've taught)

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

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

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

```
array([ 0.        , -0.05165053, -1.12617761])
```

What's happened is that we have changed `a` by changing `b`

The name `b` is bound to `a` and becomes just another reference to the array (the Python assignment model is described in more detail *later in the course*)

Hence, it has equal rights to make changes to that array

This is in fact the most sensible default behavior

It means that we pass around only pointers to data, rather than making copies

Making copies is expensive in terms of both speed and memory

Making Copies

It is of course possible to make `b` an independent copy of `a` when required

With recent versions of NumPy, this is best done using `np.copyto`

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

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

```
b = np.empty_like(a)  # empty array with same shape as a
np.copyto(b, a)  # copy to b from a
b
```

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

Now `b` is an independent copy (often 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`

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 `ValueError`

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

The NumPy function `np.where` provides a vectorized alternative:

```
import numpy as np

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 `np.vectorize` to vectorize a given function

```
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

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

```
array([ True,  True], dtype=bool)
```

```
y[0] = 5
z == y
```

```
array([False,  True], dtype=bool)
```

```
z != y
```

```
array([ True, False], dtype=bool)
```

The situation is similar for >, <, >= and <=

We can also do comparisons against scalars

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

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

```
z > 3
```

```
array([False, False,  True,  True,  True], dtype=bool)
```

This is particularly useful for *conditional extraction*

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

```
array([False, False,  True,  True,  True], dtype=bool)
```

```
z[b]
```

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

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

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

```
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*

```
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()
```

```
5.0369999999999999
```

Another commonly used subpackage is *np.linalg*

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

np.linalg.det(A)           # Compute the determinant
```

```
-2.0000000000000004
```

```
np.linalg.inv(A)           # Compute the inverse
```

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

Much of this functionality is also available in SciPy, a collection of modules that are built on top of NumPy

We'll cover the SciPy versions in more detail *soon*

For a comprehensive list of what's available in NumPy see this documentation

Exercises

Exercise 1

Consider the polynomial expression

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_N x^N = \sum_{n=0}^{N} a_n x^n \tag{1.2}$$

Earlier, you wrote a simple function `p(x, coeff)` to evaluate (1.2) 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.sum() == 1`

Suppose that `q` represents a probability mass function

We wish to generate a discrete random variable x such that $\mathbb{P}\{x = i\} = q_i$

In other words, `x` takes values in `range(len(q))` and `x = i` with probability `q[i]`

The standard (inverse transform) algorithm is as follows:

- Divide the unit interval $[0, 1]$ into n subintervals $I_0, I_1, \ldots, I_{n-1}$ 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

Solutions

Import numpy and some plotting functionality:

```
import numpy as np
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
```

Let's test it

```
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:

```
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

```
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

```
"""
Modifies ecdf.py from QuantEcon to add in a plot method

"""

class ECDF:
    """
    One-dimensional empirical distribution function given a vector of
    observations.

    Parameters
    ----------
    observations : array_like
        An array of observations

    Attributes
    ----------
    observations : array_like
        An array of observations

    """

    def __init__(self, observations):
        self.observations = np.asarray(observations)

    def __call__(self, x):
        """
        Evaluates the ecdf at x

        Parameters
        ----------
        x : scalar(float)
            The x at which the ecdf is evaluated

        Returns
        -------
        scalar(float)
            Fraction of the sample less than x

        """
        return np.mean(self.observations <= x)

    def plot(self, 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

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

Matplotlib

Contents

- *Matplotlib*

- *Overview*
- *The MATLAB-Style API*
- *The Object-Oriented API*
- *More Features*
- *Further Reading*

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
- animation, etc., etc.

Sure, you will find the odd disgruntled individual who complains about Matplotlib

Typically, this is because they don't know how to use it properly

Once you do, Matplotlib is a joy to use — powerful, elegant and intuitive

Matplotlib's Split Personality

There's one thing you need to know before we get started

Matplotlib is unusual in that it offers two quite different interfaces to plotting

The first is a simple MATLAB-style API (Application Programming Interface) that was written to help MATLAB refugees find a ready home

The second is a more "Pythonic" object-oriented API

For reasons described below, we strongly recommend that you use the second API

But first, let's discuss the difference

The MATLAB-Style API

Let's cover the MATLAB-style API first

Here's the kind of easy example you might find in introductory treatments

```
from pylab import *  # Deprecated
x = linspace(0, 10, 200)
y = sin(x)
plot(x, y, 'b-', linewidth=2)
show()
```

Typically this will appear as a separate window, like so

The buttons at the bottom of the window allow you to manipulate the figure and then save it if you wish

If you're using Jupyter notebook you can also have it appear inline, as described *here*

The `pylab` module is actually just a few lines of code instructing the interpreter to pull in some key functionality from `matplotlib` and `numpy`

It is in fact deprecated, although still in common use

Also, `from pylab import *` pulls **lots** of names into the global namespace, which is a potential source of name conflicts

A better syntax would be

```
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()
```

The Object-Oriented API

The API described above 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 (type `import this` in the IPython (or Python) shell and look at the second line)

This leads us to the alternative, object oriented Matplotlib API

Here's the code corresponding to the preceding figure using the object-oriented API

```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
x = np.linspace(0, 10, 200)
y = np.sin(x)
ax.plot(x, y, 'b-', linewidth=2)
plt.show()
```

While there's a bit more typing, the more explicit use of objects gives us more fine-grained control

This will become more clear as we go along

Incidentally, regarding the above lines of code,

- the form of the import statement `import matplotlib.pyplot as plt` is standard
- 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`

Tweaks

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

```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
```

```
x = np.linspace(0, 10, 200)
y = np.sin(x)
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')`

```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
x = np.linspace(0, 10, 200)
y = np.sin(x)
ax.plot(x, y, 'r-', linewidth=2, label='sine function', alpha=0.6)
ax.legend(loc='upper center')
plt.show()
```

If everthing is properly configured, then adding LaTeX is trivial

```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
x = np.linspace(0, 10, 200)
y = np.sin(x)
ax.plot(x, y, 'r-', linewidth=2, label=r'$y=\sin(x)$', alpha=0.6)
ax.legend(loc='upper center')
plt.show()
```

The `r` in front of the label string tells Python that this is a raw string

The figure now looks as follows

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

```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
x = np.linspace(0, 10, 200)
y = np.sin(x)
ax.plot(x, y, 'r-', linewidth=2, label=r'$y=\sin(x)$', 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

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 mutiple plots on the same axes

Here's an example that randomly generates three normal densities and adds a label with their mean

```
import matplotlib.pyplot as plt
import numpy as np
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 = r'$\mu = {0:.2f}$'.format(m)
    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

```
import matplotlib.pyplot as plt
from scipy.stats import norm
from random import uniform
num_rows, num_cols = 3, 2
fig, axes = plt.subplots(num_rows, num_cols, figsize=(8, 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 = r'$\mu = {0:.1f}, \quad \sigma = {1:.1f}$'.format(m, s)
        axes[i, j].set_title(t)
        axes[i, j].set_xticks([-4, 0, 4])
        axes[i, j].set_yticks([])
plt.show()
```

The output looks as follows

In fact the preceding figure was generated by the code above preceded by the following three lines

```
from matplotlib import rc
rc('font',**{'family':'serif','serif':['Palatino']})
rc('text', usetex=True)
```

Depending on your LaTeX installation, this may or may not work for you — try experimenting and see how you go

3D Plots

Matplotlib does a nice job of 3D plots — here is one example

The source code is

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D
import numpy as np
from matplotlib import cm


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

xgrid = np.linspace(-3, 3, 50)
ygrid = xgrid
x, y = np.meshgrid(xgrid, ygrid)

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x,
                y,
                f(x, y),
                rstride=2, cstride=2,
                cmap=cm.jet,
                alpha=0.7,
                linewidth=0.25)
ax.set_zlim(-0.5, 1.0)
plt.show()
```

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 this blog 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

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


def subplots():
    "Custom subplots with axes throught 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

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

SciPy

Contents

- *SciPy*
  - *SciPy versus NumPy*
  - *Statistics*
  - *Roots and Fixed Points*

- *Optimization*
- *Integration*
- *Linear Algebra*
- *Exercises*
- *Solutions*

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

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

```
# 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

```
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

```
from scipy.integrate import quad
from scipy.optimize import brentq
# etc
```

Let's explore some of the major subpackages

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

```
import numpy as np

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

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

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

$$f(x; a, b) = \frac{x^{(a-1)}(1-x)^{(b-1)}}{\int_0^1 u^{(a-1)}u^{(b-1)}du} \qquad (0 \leq x \leq 1) \tag{1.3}$$

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

```
import numpy as np
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()
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

```
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

```
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

```
import numpy as np
from scipy.stats import beta
import matplotlib.pyplot as plt

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

```
from scipy.stats import linregress

x = np.random.randn(200)
y = 2 * x + 0.1 * np.random.randn(200)
gradient, intercept, r_value, p_value, std_err = linregress(x, y)
gradient, intercept
```

```
(1.9962554379482236, 0.008172822032671799)
```

To see the full list, consult the documentation

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

$$f(x) = \sin(4(x - 1/4)) + x + x^{20} - 1 \tag{1.4}$$

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 it's less than 50
  - If yes, B asks if it's less than 25
  - If no, B asks if it's less than 75

And so on

This is bisection

Here's a fairly simplistic implementation of the algorithm in Python

It works for all sufficiently well behaved increasing continuous functions with $f(a) < 0 < f(b)$

```
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 it's own bisection function, which we now test using the function f defined in (1.4)

```
from scipy.optimize import bisect

f = lambda x: np.sin(4 * (x - 0.25)) + x + x**20 - 1
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

```
from scipy.optimize import newton

newton(f, 0.2)   # Start the search at initial condition x = 0.2
```

```
0.40829350427935679
```

```
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

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

```
1000 loops, best of 3: 261 us per loop
```

```
%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

```
brentq(f, 0, 1)
```

```
0.40829350427936706
```

```
%timeit brentq(f, 0, 1)
```

```
10000 loops, best of 3: 63.2 us per loop
```

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

```
from scipy.optimize import fixed_point

fixed_point(lambda x: x**2, 10.0)  # 10.0 is an initial guess
```

```
array(1.0)
```

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

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`

```
from scipy.optimize import fminbound

fminbound(lambda x: x**2, -1, 2)  # Search in [-1, 2]
```

```
0.0
```

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

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`

```
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

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

Exercises

Exercise 1

Previously we discussed the concept of *recusive function calls*

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

```
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:  np.sin(4 * (x - 0.25)) + x + x**20 - 1` discussed above

Solutions

Exercise 1

Here's a reasonable solution:

```
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('Current mid point = {}'.format(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

```
import numpy as np
f = lambda x: np.sin(4 * (x - 0.25)) + x + x**20 - 1

bisect(f, 0, 1)
```

```
Current mid point = 0.5
Current mid point = 0.25
Current mid point = 0.375
Current mid point = 0.4375
Current mid point = 0.40625
Current mid point = 0.421875
Current mid point = 0.4140625
Current mid point = 0.41015625
Current mid point = 0.408203125
Current mid point = 0.4091796875
Current mid point = 0.40869140625
Current mid point = 0.408447265625
Current mid point = 0.4083251953125
Current mid point = 0.40826416015625
```

CHAPTER

TWO

PART 2: ADVANCED PYTHON PROGRAMMING

This part provides a look at more advanced concepts in Python programming

Object Oriented Programming

Contents

- *Object Oriented Programming*
  - *Overview*
  - *About OOP*
  - *Defining Your Own Classes*
  - *Special Methods*
  - *Exercises*
  - *Solutions*

Overview

OOP is one of the major paradigms in programming, and nicely supported in Python

OOP has become an important concept in modern software engineering because

- It can help facilitate clean, efficient code (*if* used well)
- The OOP design pattern fits well with many computing problems

OOP is about producing well organized code — an important determinant of productivity

Moreover, OOP is a part of Python, and to progress further it's necessary to understand the basics

About OOP

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 look at general concepts before we specialize to Python

Key Concepts

The traditional (non-OOP) paradigm is called *procedural*, and works as follows

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

In contrast, 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.

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

```
[1, 4, 5]
```

Here `sort` is a function that is "part of" the list object

In the OOP setting, functions are usually called *methods* (e.g., `sort` is a list method)

Standard Terminology

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

```
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.__class__` are two attributes of `x`
- `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

- E.g., *a general equilibrium theory* consists of a commodity space, preferences, technologies, and an equilibrium definition
- E.g., *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

One concrete setting where OOP is almost always used is computer desktop environments with windows

Windows have common functionality and individual data, which makes them suitable for implementing with OOP

- individual data: contents of specific windows
- common functionality: closing, maximizing, etc.

Individual windows are created as objects from a class definition, with their own "instance" data

Common functionality is implemented as set of methods, which all of these objects share

Data Encapsulation

Another, more prosaic, use of OOP is data encapsulation

Data encapsulation means "hiding" variables rather than making them directly accessible

The alternative is filling the global namespace with variable names, which frequently leads to conflicts

- Think of the global namespace as any name you can refer to without a dot in front of it

Example. The modules `os` and `sys` both define a different attribute called `path`

The following code leads immediately to a conflict

```
from os import path
from sys import path
```

At this point, both variables have been brought into the global namespace, and the second will shadow the first

A better idea is to replace the above with

```
import os
import sys
```

and then reference the `path` you want with either `os.path` or `sys.path`

This example shows that modules provide one means of data encapsulation

As will now become clear, OOP provides another

Defining Your Own Classes

Let's build a super simple class as an exercise

```
class Consumer:
    pass


c1 = Consumer()  # Create an instance
c1.wealth = 10
c1.wealth
```

```
10
```

Comments on notation:

- The `class` keyword indicates that we are building a class
- The `pass` keyword is used in Python to stand in for an empty code block
- "Calling" the class with syntax `ClassName()` creates an instance of the class

Notice the flexibility of Python:

- We don't actually need to specify what attributes a class will have
- We can attach new attributes to instances of the class on the fly

However, most classes have more structure than our `Consumer` class

In fact the main point of classes is to provide a blueprint containing useful functionality for a given set of tasks

- For example, the sort method in `x.sort()` is specified in the blueprint for the list data type because it is useful for working with lists

Let's try to build something a bit closer to this standard conception of OOP

Example: Another Consumer Class

Let's build a `Consumer` class with more structure:

- A `wealth` attribute that stores the consumer's wealth (data)
- An `earn` method, where `earn(y)` increments the consumer's wealth by `y`
- A `spend` method, where `spend(x)` either decreases wealth by `x` or returns an error if insufficient funds exist

Admittedly a little contrived, this example of a class helps us internalize some new syntax

Here's one implementation, from file consumer.py

```
class Consumer:

    def __init__(self, w):
        "Initialize consumer with w dollars of wealth"
        self.wealth = w

    def earn(self, y):
        "The consumer earns y dollars"
        self.wealth += y

    def spend(self, x):
        "The consumer spends x dollars if feasible"
        new_wealth = self.wealth - x
        if new_wealth < 0:
            print("Insufficent funds")
        else:
            self.wealth = new_wealth
```

There's some special syntax here so let's step through carefully

This class defines instance data *wealth* and three methods: *__init__*, *earn* and *spend*

- *wealth* is *instance data* because each consumer we create (each instance of the *Consumer* class) will have its own separate wealth data

The ideas behind the *earn* and *spend* methods were discussed above

Both of these act on the instance data *wealth*

The *__init__* method is a *constructor method*

Whenever we create an instance of the class, this method will be called automatically

Calling *__init__* sets up a "namespace" to hold the instance data — more on this soon

We'll also discuss the role of *self* just below

Usage

Here's an example of usage

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

```
5
```

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

```
Insufficent funds
```

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

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

```
8
```

```
c1.wealth
```

```
10
```

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

```
c1.__dict__
```

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

```
c2.__dict__
```

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

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

Self

If you look at the *Consumer* class definition again you'll see the word *self* throughout the code

The rules with *self* are that

- Any instance data should be prepended with *self*
  - e.g., the *earn* method references *self.wealth* rather than just *wealth*
- Any method defined within the class should have *self* as its first argument
  - e.g., *def earn(self, y)* rather than just *def earn(y)*

- Any method referenced within the class should be called as *self.method_name*

There are no examples of the last rule in the preceding code but we will see some shortly

Details

In this section we look at some more formal details related to classes and *self*

- You might wish to skip to *the next section* on first pass of this lecture
- You can return to these details after you've familiarized yourself with more examples

Methods actually live inside a class object formed when the interpreter reads the class definition

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

```
{'earn': <function Consumer.earn at 0x7f2590054d90>,
'spend': <function Consumer.spend at 0x7f2590054e18>,
'__doc__': None,
'__weakref__': <attribute '__weakref__' of 'Consumer' objects>,
'__init__': <function Consumer.__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

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

```
20
```

When you call *earn* via *c1.earn(10)* the interpreter passes the instance *c1* and the argument *10* to *Consumer.earn*

In fact the following are equivalent

- *c1.earn(10)*
- *Consumer.earn(c1, 10)*

In the function call *Consumer.earn(c1, 10)* note that *c1* is the first argument

Recall that in the definition of the *earn* method, *self* is the first parameter

```
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

$$k_{t+1} = \frac{szk_t^\alpha + (1-d)k_t}{1+n} \tag{2.1}$$

Here

- s is an exogenously given savings rate
- z is a productivity parameter
- α is capital's share of income
- n is the population growth rate
- d is the depreciation rate

The **steady state** of the model is the k that solves (2.1) when $k_{t+1} = k_t = k$

The code shown below can be downloaded here

```
"""
Filename: solow.py
Reference: http://quant-econ.net/py/python_oop.html
"""
from __future__ import division  # Omit for Python 3.x
import numpy as np

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

    .. math::
        k_{t+1} = \frac{s z k^{\alpha}_t}{1 + n}  + k_t \frac{1 - d}{1 + n}

    """

    def __init__(self, n, s, d, alpha, z, k):
        """
        Solow growth model with Cobb Douglas production function.  All
        parameters are scalars.  See http://quant-econ.net/py/python_oop.html
        for interpretation.
        """
        self.n, self.s, self.d, self.alpha, self.z = n, s, d, alpha, z
        self.k = k


    def h(self):
        "Evaluate the h function"
        temp = self.s * self.z * self.k**self.alpha + self.k * (1 - self.d)
        return temp / (1 + self.n)
```

```
    def update(self):
        "Update the current state (i.e., the capital stock)."
        self.k =  self.h()

    def steady_state(self):
         "Compute the steady state value of capital."
         return ((self.s * self.z) / (self.n + self.d))**(1 / (1 - self.
↪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
```

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 (2.1)
- The *update* method uses *h* to update capital as per (2.1)
  - Notice how inside *update* the reference to the local method *h* is *self.h*

The methods *steady_state* and *generate_sequence* are fairly self explanatory

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

```
import matplotlib.pyplot as plt

baseline_params = 0.05, 0.25, 0.1, 0.3, 2.0, 1.0
s1 = Solow(*baseline_params)  # The 'splat' operator * breaks up the tuple
s2 = Solow(*baseline_params)
s2.k = 8.0  # Reset s2.k to make high capital economy
T = 60
fig, ax = plt.subplots()
# Plot the common steady state value of capital
ax.plot([s1.steady_state()]*T, 'k-', label='steady state')
# Plot time series for each economy
for s in s1, s2:
    lb = 'capital series from initial state {}'.format(s.k)
    ax.plot(s.generate_sequence(T), 'o-', lw=2, alpha=0.6, label=lb)

ax.legend(loc='lower right')
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 equlibrium price and quantity, tax revenue raised, consumer surplus and producer surplus

Here's our implementation

```
"""
Filename: market.py
Reference: http://quant-econ.net/py/python_oop.html
"""

from __future__ import division
from scipy.integrate import quad

class Market:
```

```
    def __init__(self, ad, bd, az, bz, tax):
        """
        Set up market parameters.  All parameters are scalars.  See
        http://quant-econ.net/py/python_oop.html for interpretation.

        """
        self.ad, self.bd, self.az, self.bz, self.tax = ad, bd, az, bz, tax
        if ad < az:
            raise ValueError('Insufficient demand.')

    def price(self):
        "Return equilibrium price"
        return  (self.ad - self.az + self.bz*self.tax)/(self.bd + self.bz)

    def quantity(self):
        "Compute equilibrium quantity"
        return  self.ad - self.bd * self.price()

    def consumer_surp(self):
        "Compute consumer surplus"
        # == Compute area under inverse demand function == #
        integrand = lambda x: (self.ad/self.bd) - (1/self.bd)* x
        area, error = quad(integrand, 0, self.quantity())
        return area - self.price() * self.quantity()

    def producer_surp(self):
        "Compute producer surplus"
        #  == Compute area above inverse supply curve, excluding tax == #
        integrand = lambda x: -(self.az/self.bz) + (1/self.bz) * x
        area, error = quad(integrand, 0, self.quantity())
        return (self.price() - self.tax) * self.quantity() - area

    def taxrev(self):
        "Compute tax revenue"
        return self.tax * self.quantity()

    def inverse_demand(self,x):
        "Compute inverse demand"
        return self.ad/self.bd - (1/self.bd)* x

    def inverse_supply(self,x):
        "Compute inverse supply curve"
        return -(self.az/self.bz) + (1/self.bz) * x + self.tax

    def inverse_supply_no_tax(self,x):
        "Compute inverse supply curve without tax"
        return -(self.az/self.bz) + (1/self.bz) * x
```

Here's a sample of usage

```
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 and curves supply with and without tax

```
import matplotlib.pyplot as plt
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

The next program provides a function that

- takes an instance of *Market* as a parameter
- computes dead weight loss from the imposition of the tax

```
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

```
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

$$x_{t+1} = r x_t (1 - x_t), \quad x_0 \in [0, 1], \quad r \in [0, 4] \tag{2.2}$$

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

Here's one implementation, in file chaos_class.py

```
"""
Filename: chaos_class.py
Reference: http://quant-econ.net/py/python_oop.html
"""
class Chaos:
    """
    Models the dynamical system with :math:`x_{t+1} = r x_t (1 - x_t)`
    """
    def __init__(self, x0, r):
        """
        Initialize with state x0 and parameter r
        """
        self.x, self.r = x0, r

    def update(self):
        "Apply the map to update state."
        self.x =  self.r * self.x *(1 - self.x)

    def generate_sequence(self, n):
        "Generate and return a sequence of length n."
        path = []
        for i in range(n):
            path.append(self.x)
            self.update()
        return path
```

Here's an example of usage

```
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

```
"""
Filename: choatic_ts.py
Reference: http://quant-econ.net/py/python_oop.html
"""
import matplotlib.pyplot as plt

ch = Chaos(0.1, 4.0)
ts_length = 250

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

```
ax.set_xlabel(r'$t$', fontsize=14)
ax.set_ylabel(r'$x_t$', fontsize=14)
x = ch.generate_sequence(ts_length)
ax.plot(range(ts_length), x, 'bo-', alpha=0.5, lw=2, label=r'$x_t$')
plt.show()
```

The resulting figure looks as follows

The next piece of code provides a bifurcation diagram

```
"""
Filename: bifurcation_diagram.py
Reference: http://quant-econ.net/py/python_oop.html
"""
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ch = Chaos(0.1, 4)
r = 2.5
while r < 4:
    ch.r = r
    t = ch.generate_sequence(1000)[950:]
    ax.plot([r] * len(t), t, 'b.', ms=0.6)
    r = r + 0.005

ax.set_xlabel(r'$r$', fontsize=16)
plt.show()
```

Here is the figure it generates

On the horizontal axis is the parameter r in (2.2)

The vertical axis is the state space $[0, 1]$

For each r we compute a long time series and then plot the tail (the last 50 points)

The tail of the sequence shows us where the trajectory concentrates after settling down to some kind of steady state, if a steady state exists

Whether it settles down, and the character of the steady state to which it does settle down, depend on the value of r

For r between about 2.5 and 3, the time series settles into a single fixed point plotted on the vertical axis

For r between about 3 and 3.45, the time series settles down to oscillating between the two values plotted on the vertical axis

For r a little bit higher than 3.45, the time series settles down to oscillating among the four values plotted on the vertical axis

Notice that there is no value of r that leads to a steady state oscillating among three values

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

```
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

```
class Foo:

    def __len__(self):
        return 42
```

Now we get

```
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

```
class Foo:

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

After running we get

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

```
50
```

Exercise 1 provides a more useful example

Exercises

Exercise 1

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

$$F_n(x) := \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\{X_i \leq x\} \qquad (x \in \mathbb{R}) \tag{2.3}$$

Here $\mathbf{1}\{X_i \leq x\}$ is an indicator function (one if $X_i \leq x$ and zero otherwise) and hence $F_n(x)$ is the fraction of the sample that falls below x

The Glivenko–Cantelli Theorem states that, provided that the sample is iid, the ecdf F_n converges to the true distribution function F

Implement F_n as a class called `ECDF`, where

- A given sample $\{X_i\}_{i=1}^n$ are the instance data, stored as `self.observations`
- The class implements a `__call__` method that returns $F_n(x)$ for any x

Your code should work as follows (modulo randomness)

```
from random import uniform

samples = [uniform(0, 1) for i in range(10)]
F = ECDF(samples)
F(0.5)  # Evaluate ecdf at x = 0.5
```

```
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 `Polynomial` for representing and manipulating polynomial functions such as

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_N x^N = \sum_{n=0}^{N} a_n x^n \qquad (x \in \mathbb{R}) \tag{2.4}$$

The instance data for the class `Polynomial` will be the coefficients (in the case of (2.4), the numbers $a_0, \ldots, a_N$)

Provide methods that

1. Evaluate the polynomial (2.4), returning $p(x)$ for any x
2. Differentiate the polynomial, replacing the original coefficients with those of its derivative p'

Avoid using any `import` statements

Solutions

Exercise 1

```
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

```
class Polynomial:

    def __init__(self, coefficients):
        """
        Creates an instance of the Polynomial class representing

            p(x) = a_0 x^0 + ... + a_N x^N,

        where a_i = coefficients[i].
        """
        self.coefficients = coefficients

    def __call__(self, x):
        "Evaluate the polynomial at x."
        y = 0
        for i, a in enumerate(self.coefficients):
            y += a * x**i
```

```
        return y

    def differentiate(self):
        "Reset self.coefficients to those of p' instead of p."
        new_coefficients = []
        for i, a in enumerate(self.coefficients):
            new_coefficients.append(i * a)
        # Remove the first element, which is zero
        del new_coefficients[0]
        # And reset coefficients data to new values
        self.coefficients = new_coefficients
        return new_coefficients
```

How it Works: Data, Variables and Names

Contents

- *How it Works: Data, Variables and Names*
  - *Overview*
  - *Objects*
  - *Iterables and Iterators*
  - *Names and Name Resolution*

Overview

The objective of the lecture is to provide deeper understanding of Python's execution model

Understanding these details is important for writing larger programs

You should feel free to **skip this material on first pass** and continue on to the applications

We provide this material mainly as a reference, and for returning to occasionally to build your Python skills

Objects

We discussed objects briefly in *the previous lecture*

Objects are usually thought of as instances of some class definition, typically combining both data and methods (functions)

For example

```
x = ['foo', 'bar']
```

creates (an instance of) a list, possessing various methods (`append`, `pop`, etc.)

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.

At this point it is helpful to have a clearer idea of what an object is in Python

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

1. a type
2. some content
3. a unique identity
4. zero or more methods

These concepts are discussed sequentially in the remainder of this section

Type

Python understands and provides for different types of objects, to accommodate different types of data

The type of an object can be queried via `type(object_name)`

For example

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

```
str
```

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

```
int
```

The type of an object matters for many expressions

For example, the addition operator between two strings means concatenation

```
'300' + 'cc'
```

```
'300cc'
```

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

```
300 + 400
```

```
700
```

Consider the following expression

```
'300' + 400
```

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

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

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

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-9b7dffd27f2d> in <module>()
----> 1 '300' + 400

TypeError: Can't convert 'int' object to str implicitly
```

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

For example,

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

```
700
```

Content

The content of an object seems like an obvious concept

For example, if we set `x = 42` then it might seem that the content of `x` is just the number 42

But actually, there's more, as the following example shows

```
x = 42
x
```

```
42
```

```
x.imag
```

```
0
```

```
x.__class__
```

```
int
```

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

As discussed *previously*, any name following a dot is called an *attribute* of the object to the left of the dot

- For example, `imag` and `__class__` are attributes of `x`

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

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

```
166719660
```

```
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

Methods

As discussed earlier, methods are functions that are bundled with objects

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

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

```
True
```

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

```
False
```

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

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

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

```
s.lower()
```

```
'this is a string'
```

```
s.replace('This', 'That')
```

```
'That is a string'
```

A great deal of Python functionality is organized around method calls

For example, consider the following piece of code

```
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

```
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)

Everything is an Object

Above we said that in Python everything is an object—let's look at this again

Consider, for example, functions

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

The following code illustrates

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

```
<function __main__.f>
```

```
type(f)
```

```
function
```

```
id(f)
```

```
3074342220L
```

```
f.__name__
```

```
'f'
```

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

Likewise modules loaded into memory are treated as objects

```
import math

id(math)
```

```
3074329380L
```

This uniform treatment of data in Python (everything is an object) helps keep the language simple and consistent

Iterables and Iterators

We've *already said something* about iterating in Python

Now let's look more closely at how it all works, focusing in Python's implementation of the `for` loop

Iterators

Iterators are a uniform interface to stepping through elements in a collection

Here we'll talk about using iterators—later we'll learn how to build our own

Formally, an *iterator* is an object with a `__next__` method

For example, file objects are iterators

To see this, let's have another look at the *US cities data*

```
f = open('us_cities.txt')
f.__next__()
```

```
'new york: 8244910\n'
```

```
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

```
next(f)
```

```
'chicago: 2707120 \n'
```

The objects returned by `enumerate()` are also iterators

```
e = enumerate(['foo', 'bar'])
next(e)
```

```
(0, 'foo')
```

```
next(e)
```

```
(1, 'bar')
```

as are the reader objects from the `csv` module

```
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']
```

```
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

```
for x in iterator:
    <code block>
```

then the interpreter

- calls `iterator.__next__()` and binds `x` to the result
- executes the code block
- repeats until a `StopIteration` error occurs

So now you know how this magical looking syntax works

```
f = open('somefile.txt', 'r')
for line in f:
    # do something
```

The interpreter just keeps

1. calling `f.__next__()` and binding `line` to the result
2. executing the body of the loop

This continues until a `StopIteration` error occurs

Iterables

You already know that we can put a Python list to the right of `in` in a `for` loop

```
for i in ['spam', 'eggs']:
    print(i)
```

```
spam
eggs
```

So does that mean that a list is an iterator?

The answer is no:

```
type(x)
```

```
list
```

```
next(x)
```

```
---------------------------------------------------------------------------
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 *iterable* (as opposed to an iterator)

Formally, an object is iterable if it can be converted to an iterator using the built-in function `iter()`

Lists are one such object

```
x = ['foo', 'bar']
type(x)
```

```
list
```

```
y = iter(x)
type(y)
```

```
list_iterator
```

```
next(y)
```

```
'foo'
```

```
next(y)
```

```
'bar'
```

```
next(y)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-62-75a92ee8313a> in <module>()
----> 1 y.next()

StopIteration:
```

Many other objects are iterable, such as dictionaries and tuples

Of course, not all objects are iterable

```
iter(42)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-63-826bbd6e91fc> in <module>()
----> 1 iter(42)

TypeError: 'int' object is not iterable
```

To conclude our discussion of `for` loops

- `for` loops work on either iterators or iterables

- In the second case, the iterable is converted into an iterator before the loop starts

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]
max(x)
```

```
10
```

```
y = iter(x)
type(y)
```

```
listiterator
```

```
max(y)
```

```
10
```

One thing to remember about iterators is that they are depleted by use

```
x = [10, -10]
y = iter(x)
max(y)
```

```
10
```

```
max(y)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-72-1d3b6314f310> in <module>()
----> 1 max(y)

ValueError: max() arg is an empty sequence
```

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 `int` in your computer's memory, containing

- the value `42`
- some associated attributes

But what is `x` itself?

In Python, `x` is called a *name*, and the statement `x = 42` *binds* the name `x` to the integer object we have just discussed

Under the hood, this process of binding names to objects is implemented as a dictionary—more about this in a moment

There is no problem binding two or more names to the one object, regardless of what that object is

```
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

```
x = 'foo'
id(x)
```

```
164994764
```

```
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

```
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

```
# Filename: math2.py
pi = 'foobar'
```

Now we start the Python interpreter and import it

```
import math2
```

Next let's import the `math` module from the standard library

```
import math
```

Both of these modules have an attribute called `pi`

```
math.pi
```

```
3.1415926535897931
```

```
math2.pi
```

```
'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__`

```
import math

math.__dict__
```

```
{'pow': <built-in function pow>, ..., 'pi': 3.1415926535897931,...}  # Edited␣
↪output
```

```
import math2

math2.__dict__
```

```
{..., '__file__': 'math2.py', 'pi': 'foobar',...}  # Edited output
```

As you know, we access elements of the namespace using the dotted attribute notation

```
math.pi
```

```
3.1415926535897931
```

In fact this is entirely equivalent to `math.__dict__['pi']`

```
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)`

```
vars(math)
```

```
{'pow': <built-in function pow>,...
```

If you just want to see the names, you can type

```
dir(math)
```

```
['__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

```
print(math.__doc__)
```

```
This module is always available.  It provides access to the
mathematical functions defined by the C standard.
```

```
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

```
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

```
# Filename: mod.py
print(__name__)
```

Now let's look at two different ways of running it in IPython

```
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`

```
x = 2
y = 3

import numpy as np

whos
```

```
Variable   Type      Data/Info
------------------------------
np         module    <module 'numpy' from '/us<...>ages/numpy/__init__.pyc'>
x          int       2
y          int       3
```

The Global Namespace

Python documentation often makes reference to the "global namespace"

The global namespace is *the namespace of the module currently being executed*

For example, suppose that we start the interpreter and begin making assignments

We are now working in the module `__main__`, and hence the namespace for `__main__` is the global namespace

Next, we import a module called `amodule`

```
import amodule
```

At this point, the interpreter creates a namespace for the module `amodule` and starts executing commands in the module

While this occurs, the namespace `amodule.__dict__` is the global namespace

Once execution of the module finishes, the interpreter returns to the module from where the import statement was made

In this case it's `__main__`, so the namespace of `__main__` again becomes the global namespace

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

```
def f(x):
    a = 2
    print(locals())
    return a * x
```

Now let's call the function

```
f(1)
```

```
{'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?

- 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 that's printed)

However, we do need to understand how the Python interpreter works with multiple namespaces

At any point of execution, there are in fact at least two namespaces that can be accessed directly

("Accessed directly" means without using a dot, as in `pi` rather than `math.pi`)

These namespaces are

- The global namespace (of the module being executed)
- The builtin namespace

If the interpreter is executing a function, then the directly accessible namespaces are

- The local namespace of the function
- The global namespace (of the module being executed)
- The builtin namespace

Sometimes functions are defined within other functions, like so

```
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

```
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
```

```
x
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-401b30e3b8b5> in <module>()
----> 1 x

NameError: name 'x' is not defined
```

First,

- The global namespace `{}` is created
- The function object is created, and `g` is bound to it within the global namespace
- The name `a` is bound to `0`, again in the global namespace

Next `g` is called via `y = g(10)`, leading to the following sequence of actions

- The local namespace for the function is created
- Local names `x` and `a` are bound, so that the local namespace becomes `{'x': 10, 'a': 1}`
- Statement `x = x + a` uses the local `a` and local `x` to compute `x + a`, and binds local name `x` to the result
- This value is returned, and `y` is bound to it in the global namespace
- Local `x` and `a` are discarded (and the local namespace is deallocated)

Note that the global `a` was not affected by the local `a`

Mutable Versus Immutable Parameters

This is a good time to say a little more about mutable vs immutable objects

Consider the code segment

```
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, it's a different story when we use a **mutable** data type such as a list

```
def f(x):
    x[0] = x[0] + 1
    return x

x = [1]
print(f(x), x)
```

This prints `[2]` as the value of `f(x)` and *same* for `x`

Here's what happens

- `f` is registered as a function in the global namespace
- `x` bound to `[1]` in the global namespace
- The call `f(x)`
  - Creates a local namespace
  - Adds `x` to local namespace, bound to `[1]`
  - The list `[1]` is modified to `[2]`
  - Returns the list `[2]`
  - The local namespace is deallocated, and local `x` is lost
- Global `x` has been modified

More Language Features

Contents

- *More Language Features*
  - *Overview*
  - *Handling Errors*
  - *Decorators and Descriptors*
  - *Generators*
  - *Recursive Function Calls*
  - *Exercises*
  - *Solutions*

Overview

As with the last lecture, our advice is to **skip this lecture on first pass**, unless you have a burning desire to read it

It's here

1. as a reference, so we can link back to it when required, and
2. for those who have worked through a number of applications, and now want to learn more about the Python language

A variety of topics are treated in the lecture, including generators, exceptions and descriptors

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

$$s^2 := \frac{1}{n-1} \sum_{i=1}^{n} (y_i - \bar{y})^2 \qquad \bar{y} = \text{ sample mean}$$

This can be calculated in NumPy using `np.var`

But if you were writing a function to handle such a calculation, you might anticipate a divide-by-zero error when the sample size is one

One possible action is to do nothing — the program will just crash, and spit out an error message

But sometimes it's worth writing your code in a way that anticipates and deals with runtime errors that you think might arise

Why?

- Because the debugging information provided by the interpreter is often less useful than the information on possible errors you have in your head when writing code
- Because errors causing execution to stop are frustrating if you're in the middle of a large computation
- Because it's reduces confidence in your code on the part of your users (if you are writing for others)

Assertions

A relatively easy way to handle checks is with the `assert` keyword

For example, pretend for a moment that the `np.var` function doesn't exist and we need to write our own

```
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

```
var([1])
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-20-0032ff8a150f> in <module>()
----> 1 var([1])

<ipython-input-19-cefafaec3555> in var(y)
      1 def var(y):
      2     n = len(y)
----> 3     assert n > 1, 'Sample size must be greater than one.'
      4     return np.sum((y - y.mean())**2) / float(n-1)

AssertionError: Sample size must be greater than one.
```

The advantage is that we can

- fail early, as soon as we know there will be a problem
- supply specific information on why a program is failing

Handling Errors During Runtime

The approach used above is a bit limited, because it always leads to termination

Sometimes we can handle errors more gracefully, by treating special cases

Let's look at how this is done

Exceptions

Here's an example of a common error type

```
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

Here's a different kind of error, unrelated to syntax

```
1 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-17-05c9758a9c21> in <module>()
----> 1 1/0
```

```

ZeroDivisionError: integer division or modulo by zero
```

Here's another

```
x1 = y1
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-23-142e0509fbd6> in <module>()
----> 1 x1 = y1

NameError: name 'y1' is not defined
```

And another

```
'foo' + 6
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-44bbe7e963e7> in <module>()
----> 1 'foo' + 6

TypeError: cannot concatenate 'str' and 'int' objects
```

And another

```
X = []
x = X[0]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-22-018da6d9fc14> in <module>()
----> 1 x = X[0]

IndexError: list index out of range
```

On each occaision, 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

```
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

```
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

```
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

```
f(2)
```

```
0.5
```

```
f(0)
```

```
Error: Division by zero.  Returned None
```

```
f('foo')
```

```
Error: Unsupported operation.  Returned None
```

If we feel lazy we can catch these errors together

```
def f(x):
    try:
        return 1.0 / x
    except (TypeError, ZeroDivisionError):
        print('Error: Unsupported operation.  Returned None')
    return None
```

Here's what happens

```
f(2)
```

```
0.5
```

```
f(0)
```

```
Error: Unsupported operation.  Returned None
```

```
f('foo')
```

```
Error: Unsupported operation.  Returned None
```

If we feel extra lazy we can catch all error types as follows

```
def f(x):
    try:
        return 1.0 / x
    except:
        print('Error.  Returned None')
    return None
```

In general it's better to be specific

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

```
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

```
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

```
import numpy as np

def check_nonneg(func):
    def safe_function(x):
        assert x >= 0, "Argument must be nonnegative"
        return func(x)
    return safe_function

def f(x):
    return np.log(np.log(x))

def g(x):
    return np.sqrt(42 * x)

f = check_nonneg(f)
g = check_nonneg(g)
# Program continues with various calculations using f and g
```

This looks complicated so let's work through it slowly

To unravel the logic, consider what happens when we say `f = check_nonneg(f)`

This calls the function `check_nonneg` with parameter `func` set equal to `f`

Now `check_nonneg` creates a new function called `safe_function` that verifies `x` as 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

```
def f(x):
    return np.log(np.log(x))
```

They may well miss the line `f = check_nonneg(f)`

For this and other reasons, decorators were introduced to Python

With decorators, we can replace the lines

```
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

```
@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

```
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

```
car = Car()
car.miles
```

```
1000
```

```
car.kms
```

```
1610.0
```

```
car.miles = 6000
car.kms
```

```
1610.0
```

In the last two lines we see that `miles` and `kms` are out of sync

What we really want is some mechanism whereby each time a user sets one of these variables, *the other is automatically updated*

A Solution

In Python, this issue is solved using *descriptors*

A descriptor is just a Python object that implements certain methods

These methods are triggered when the object is accessed through dotted attribute notation

The best way to understand this is to see it in action

Consider this alternative version of the `Car` class

```
class Car:

    def __init__(self, miles=1000):
        self._miles = miles
        self._kms = miles * 1.61

    def set_miles(self, value):
        self._miles = value
        self._kms = value * 1.61

    def set_kms(self, value):
        self._kms = value
        self._miles = value / 1.61

    def get_miles(self):
        return self._miles

    def get_kms(self):
        return self._kms

    miles = property(get_miles, set_miles)
    kms = property(get_kms, set_kms)
```

First let's check that we get the desired behavior

```
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 builtin Python function `property` takes getter and setter methods and creates a property

For example, after `car` is created as an instance of `Car`, the object `car.miles` is a property

Being a property, when we set its value via `car.miles = 6000` its setter method is triggered — in this case `set_miles`

Decorators and Properties

These days its very common to see the `property` function used via a decorator

Here's another version of our `Car` class that works as before but now uses decorators to set up the properties

```
class Car:

    def __init__(self, miles=1000):
        self._miles = miles
        self._kms = miles * 1.61

    @property
    def miles(self):
        return self._miles

    @property
    def kms(self):
        return self._kms
```

```
    @miles.setter
    def miles(self, value):
        self._miles = value
        self._kms = value * 1.61

    @kms.setter
    def kms(self, value):
        self._kms = value
        self._miles = value / 1.61
```

We won't go through all the details here

For further information you can refer to the descriptor documentation

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:

```
singular = ('dog', 'cat', 'bird')
type(singular)
```

```
tuple
```

```
plural = [string + 's' for string in singular]
plural
```

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

```
type(plural)
```

```
list
```

And here is the generator expression

```
singular = ('dog', 'cat', 'bird')
plural = (string + 's' for string in singular)
type(plural)
```

```
generator
```

```
next(plural)
```

```
'dogs'
```

```
next(plural)
```

```
'cats'
```

```
next(plural)
```

```
'birds'
```

Since `sum()` can be called on iterators, we can do this

```
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

```
sum(x * x for x in range(10))
```

```
285
```

Generator Functions

The most flexible way to create generator objects is to use generator functions

Let's look at some examples

Example 1

Here's a very simple example of a generator function

```
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

```
type(f)
```

```
function
```

```
gen = f()
gen
```

```
<generator object f at 0x3b66a50>
```

```
next(gen)
```

```
'start'
```

```
next(gen)
```

```
'middle'
```

```
next(gen)
```

```
'end'
```

```
next(gen)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-21-b2c61ce5e131> in <module>()
----> 1 gen.next()

StopIteration:
```

The generator function `f()` is used to create generator objects (in this case `gen`)

Generators are iterators, because they support a `next` method

The first call to `next(gen)`

- Executes code in the body of `f()` until it meets a `yield` statement
- Returns that value to the caller of `next(gen)`

The second call to `next(gen)` starts executing *from the next line*

```
def f():
    yield 'start'
    yield 'middle'  # This line!
    yield 'end'
```

and continues until the next `yield` statement

At that point it returns the value following `yield` to the caller of `next(gen)`, and so on

When the code block ends, the generator throws a `StopIteration` error

Example 2

Our next example receives an argument `x` from the caller

```
def g(x):
    while x < 100:
        yield x
        x = x * x
```

Let's see how it works

```
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*

```
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

```
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

```
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

```
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)]
```

We can avoid these problems using iterators

Here is the generator function

```
import random
def f(n):
    i = 1
    while i <= n:
```

```
        yield random.uniform(0, 1) < 0.5
        i += 1
```

Now let's do the sum

```
n = 10000000
draws = f(n)
draws
```

```
<generator object at 0xb7d8b2cc>
```

```
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

Recursive Function Calls

This is not something that you will use every day, but it is still useful — you should learn it at some stage

Basically, a recursive function is a function that calls itself

For example, consider the problem of computing x_t for some t when

$$x_{t+1} = 2x_t, \quad x_0 = 1 \tag{2.5}$$

Obviously the answer is 2^t

We can compute this easily enough with a loop

```
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

```
def x(t):
    if t == 0:
        return 1
    else:
        return 2 * x(t-1)
```

What happens here is that each successive call uses it's own *frame* in the *stack*

- a frame is where the local variables of a given function call are held

- **stack is memory used to process function calls**
  - a First In Last Out (FILO) queue

This example is somewhat contrived, since the first (iterative) solution would usually be preferred to the recursive solution

We'll meet less contrived applications of recursion later on

Exercises

Exercise 1

The Fibonacci numbers are defined by

$$x_{t+1} = x_t + x_{t-1}, \quad x_0 = 0, \; x_1 = 1 \tag{2.6}$$

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 you've put in your current working directory

```
def column_iterator(target_file, column_number):
    """A generator function for CSV files.
    When called with a file name target_file (string) and column number
    column_number (integer), the generator function returns a generator
    that steps through the elements of column column_number in file
    target_file.
    """
    # put your code here

dates = column_iterator('test_table.csv', 1)

for date in dates:
    print(date)
```

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

Solutions

Exercise 1

Here's the standard solution

```
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

```
print([x(i) for i in range(10)])
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Exercise 2

One solution is as follows

```
def column_iterator(target_file, column_number):
    """A generator function for CSV files.
    When called with a file name target_file (string) and column number
    column_number (integer), the generator function returns a generator
    which steps through the elements of column column_number in file
    target_file.
    """
    f = open(target_file, 'r')
    for line in f:
        yield line.split(',')[column_number - 1]
    f.close()

dates = column_iterator('test_table.csv', 1)

i = 1
for date in dates:
    print(date)
    if i == 10:
        break
    i += 1
```

```
Date
2009-05-21
2009-05-20
2009-05-19
2009-05-18
2009-05-15
2009-05-14
2009-05-13
2009-05-12
2009-05-11
```

Exercise 3

Let's save the data first

```
%%file numbers.txt
prices
3
8

7
21
```

```
Writing numbers.txt
```

```
f = open('numbers.txt')

total = 0.0
for line in f:
    try:
        total += float(line)
    except ValueError:
        pass

f.close()

print(total)
```

```
39.0
```

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

Overview

Are you one of those programmers who fills their code with `print` statements when trying to debug their programs?

Hey, we all used to do that

(OK, sometimes we still do that...)

But once you start writing larger programs you'll need a better system

Debugging tools for Python vary across platforms, IDEs and editors

Here we'll focus on Jupyter and leave you to explore other settings

Debugging

The `debug` Magic

Let's consider a simple (and rather contrived) example, where we have a script called `temp.py` with the following contents:

```
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)
<ipython-input-2-645aedc8a285> in <module>()
----> 1 ax

NameError: name 'ax' is not defined
```

The problem is that `ax` was defined inside `plot_log()`, and the name is lost once that function terminates

Let's try doing it a different way

We run the first cell block again, generating the same error

```
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                            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'
```

But this time we type in the following cell block

```
%debug
```

You should be dropped into a new prompt that looks something like this

```
ipdb>
```

(You might see *pdb*> instead)

Now we can investigate the value of our variables at this point in the program, step forward through the code, etc.

For example, here we simply type the name `ax` to see what's happening with this object:

```
ipdb> ax
array([<matplotlib.axes.AxesSubplot object at 0x290f5d0>,
       <matplotlib.axes.AxesSubplot object at 0x2930810>], dtype=object)
```

It's now very clear that `ax` is an array, which clarifies the source of the problem

To find out what else you can do from inside `ipdb` (or `pdb`), use the online help

```
ipdb> h

Documented commands (type help <topic>):
========================================
EOF    bt         cont      enable  jump  pdef   r        tbreak   w
a      c          continue  exit    l     pdoc   restart  u        whatis
alias  cl         d         h       list  pinfo  return   unalias  where
args   clear      debug     help    n     pp     run      unt
b      commands   disable   ignore  next  q      s        until
break  condition  down      j       p     quit   step     up

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
retval  rv
```

```
ipdb> h c
c(ont(inue))
Continue execution, only stop when a breakpoint is encountered.
```

Setting a Break Point

The preceding approach is handy but sometimes insufficient

For example, consider the following modified version of `temp.py`:

```
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 we've accidentally written `np.logspace(1, 2, 10)` instead of `np.linspace(1, 2, 10)`

Now there won't be any exception, but the plot won't look right

To investigate, it would be helpful if we could inspect variables like `x` during execution of the function

To this end , we add a "break point" by inserting the line `from IPython.core.debugger import Tracer; Tracer()()` inside the function code block

```
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

```
> <ipython-input-5-c5864f6d184b>(6)plot_log()
      4 def plot_log():
      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))

ipdb> n
> <ipython-input-5-c5864f6d184b>(7)plot_log()
      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
> <ipython-input-5-c5864f6d184b>(8)plot_log()
      6     fig, ax = plt.subplots()
      7     x = np.logspace(1, 2, 10)
----> 8     ax.plot(x, np.log(x))
      9     plt.show()
     10

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`

Other Useful Magics

In this lecture we used the `%debug` IPython magic

There are many other useful magics:

- `%precision 4` sets printed precision for floats to 4 decimal places
- `%whos` gives a list of variables and their values
- `%quickref` gives a list of magics

The full list of magics is here

Writing Good Code

Contents

- *Writing Good Code*
  - *Overview*

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

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

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

$$k_{t+1} = sk_t^\alpha + (1-\delta)k_t, \quad t = 0, 1, 2, \ldots \tag{2.7}$$

Here

- k_t is capital at time t and
- s, α, δ are parameters (savings, a productivity parameter and depreciation)

For each parameterization, the code

1. sets $k_0 = 1$
2. iterates using (2.7) 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

Here's the figure that we produce

And here's some code that produces it

```
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=(9, 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]
    label = "$\\alpha = {},\; s = {},\; \\delta={}$"
    axes[0].plot(k, 'o-', label=label.format(alpha[j], s, delta))

axes[0].grid(lw=0.2)
axes[0].set_ylim(0, 18)
axes[0].set_xlabel('time')
axes[0].set_ylabel('capital')
axes[0].legend(loc='upper left', frameon=True, fontsize=14)

# Trajectories with different s
```

```
delta = 0.1
alpha = 0.33
s = (0.3, 0.4, 0.5)

for j in range(3):
    k[0] = 1
    for t in range(49):
        k[t+1] = s[j] * k[t]**alpha + (1 - delta) * k[t]
    label = "$\\alpha = {},\; s = {},\; \\delta={}$"
    axes[1].plot(k, 'o-', label=label.format(alpha, s[j], 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]**alpha + (1 - delta[j]) * k[t]
    label = "$\\alpha = {},\; s = {},\; \\delta={}$"
    axes[2].plot(k, 'o-', label=label.format(alpha, s, 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, it's very poorly structured

Let's talk about why that's the case, and what we can do about it

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

We'll 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'll discuss how 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'll 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'll learn more about these topics over time

(Personal preference is part of the story too)

What's really important is that you use one or the other or both

Revisiting the Example

Here's 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

```
import numpy as np
import matplotlib.pyplot as plt
from itertools import product

def plot_path(ax, alphas, s_vals, deltas, series_length=50):
    """
    Add a time series plot to the axes ax for all given parameters.
    """
    k = np.empty(series_length)
    label = "$\\alpha = {},\; s = {},\; \\delta = {}$"

    for (alpha, s, delta) in product(alphas, s_vals, deltas):
        k[0] = 1
        for t in range(series_length-1):
            k[t+1] = s * k[t]**alpha + (1 - delta) * k[t]
        ax.plot(k, 'o-', label=label.format(alpha, s, delta))

    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=(9, 15))

# Parameters (alphas, s_vals, deltas)
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)):
    alphas, s_vals, deltas = params
    plot_path(ax, alphas, s_vals, deltas)

plt.show()
```

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

The Need for Speed

Contents

- *The Need for Speed*
  - *Overview*
  - *Where are the Bottlenecks?*
  - *Vectorization*
  - *Numba*
  - *Cython*
  - *Caching*
  - *Other Options*
  - *Exercises*
  - *Solutions*

Overview

Higher level languages such as Python are optimized for humans

This means that the programmer can leave many details to the runtime environment

- specifying variable types
- memory allocation/deallocation, etc.

The upside is that, compared to low-level languages, Python is typically faster to write, less error prone and easier to debug

The downside is that Python is harder to optimize — that is, turn into fast machine code — than languages like C or Fortran

Indeed, the standard implementation of Python (called CPython) cannot match the speed of compiled languages such as C or Fortran

Does that mean that we should just switch to C or Fortran for everything?

The answer is no, no and one hundred times no

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 with minor modifications

This lecture will walk you through some of the most popular options for implementing this last step

(A number of other useful options are mentioned *below*)

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

```
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

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

```
'foobar'
```

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

```
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

```
#include <stdio.h>

int main(void) {
    int i;
    int sum = 0;
    for (i = 1; i <= 10; i++) {
        sum = sum + i;
    }
    printf("sum = %d\n", sum);
    return 0;
}
```

The variables `i` and `sum` are explicitly declared to be integers

Hence, the meaning of addition here is completely unambiguous

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

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

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

Now let's try this non-vectorized code

```
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

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

```
TOC: Elapsed: 0.0016531944274902344 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*

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:

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

```
0.54030230586813977
```

```
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 let's choose

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

Here's a plot of f

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

```
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
for x in grid:
    for y in grid:
        z = f(x, y)
        if z > m:
            m = z
print(m)
```

And here's a vectorized version

```
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)
print(np.max(f(x, y)))
```

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

If you time these code snippets you will see that the second version is much faster — about two orders of magnitude

Pros and Cons of Vectorization

At its best, vectorization yields fast, simple code

However, it's not without disadvantages

One issue is that it can be highly memory intensive

For example, the vectorized maximization routine above is far more memory intensive than the non-vectorized version that preceded it

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

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 isn't 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) / cmd 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

First let's import Numba's `jit` function

```
from numba import jit
```

Now, here's a function to generate a trajectory of a given length from a given initial condition

```
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
```

To speed this up using Numba is trivial

```
qm_numba = jit(qm)  # qm_numba is now a 'compiled' version of qm
```

Let's time and compare identical function calls across these two versions:

```
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:

```
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 "numbafied" version of `qm`, you can just put `@jit` before the function

```
@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, it's prudent when using Numba to focus on speeding up small, time-critical snippets of code

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

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 extentions — 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 α, 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's 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

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

```
TOC: Elapsed: 0.11026620864868164 seconds.
```

```
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 implemention that initializes, populates and returns a NumPy array

```
%%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 it's performance is disappointing — nothing like the speed gain we got from Numba

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

```
TOC: Elapsed: 0.03628730773925781 seconds.
```

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

```
TOC: Elapsed: 0.0003917217254638672 seconds.
```

The reason is that working with NumPy arrays incurs substantial Python overheads

We can do better by using Cython's 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()
```

```
TOC: Elapsed: 0.0007178783416748047 seconds.
```

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,

- Cython is a very mature, stable and widely used tool
- Cython can be more useful than Numba when working with larger, more sophisticated applications

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

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

This is a more traditional and generic way to speed up code that can nonetheless be very useful for economic modeling

Joblib

Our caching will use the joblib library, which you need to install to run the code below

This can be done at a terminal / cmd prompt by typing

```
pip install joblib
```

An Example

Let's look at a toy example, related to the quadratic map model discussed *above*

Let's 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

(We'll omit JIT compilation or other speed ups for simplicity)

Here's our code

```
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 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

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

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

TOC: Elapsed: 6.922961473464966 seconds.
```

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

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

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

Other Options

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

We mention only a few of the most popular methods

Interfacing with Fortran

If you are comfortable writing Fortran you will find it very easy to create extention 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

Parallel and Cloud Computing

This is a big topic that we won't address in detail yet

However, you might find the following links a useful starting point

- IPython for parallel computing
- NumbaPro
- The Starcluster interface to Amazon's EC2
- Anaconda Accelerate

Exercises

Exercise 1

Later we'll learn all about *finite state Markov chains*

For now, let's just concentrate on simulating a very simple example of such a chain

Suppose that the volatility of returns on an asset can be in one of two regimes — high or low

The transition probabilities across states are as follows

For example, let the period length be one 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

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

```
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:
            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

```
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

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

```
TOC: Elapsed: 0.07770729064941406 seconds.
```

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

```
compute_series_numba = jit(compute_series)
```

Let's check we still get the right numbers

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

```
0.66764
```

Let's see the time

```
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

```
%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

Appendix — Other Options

There are other important projects aimed at speeding up Python

These include but are not limited to

- Pythran : A Python to C++ compiler
- PyPy : Runtime environment using just-in-time compiler
- Nuitka : Another Python compiler
- Pyjion : JIT compilation, under development

CHAPTER

THREE

PART 3: DATA AND EMPIRICS

This part of the course provides a set of lectures focused on Data and Empirics using Python

Pandas

Contents

- *Pandas*
  - *Overview*
  - *Series*
  - *DataFrames*
  - *On-Line Data Sources*
  - *Exercises*
  - *Solutions*

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.

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*

```
s = pd.Series(np.random.randn(4), name='daily returns')
s
```

```
0    0.430271
1    0.617328
2   -0.265421
3   -0.836113
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

```
s * 100
```

```
0    43.027108
1    61.732829
2   -26.542104
3   -83.611339
Name: daily returns, dtype: float64
```

```
np.abs(s)
```

```
0    0.430271
1    0.617328
2    0.265421
3    0.836113
Name: daily returns, dtype: float64
```

But `Series` provide more than NumPy arrays

Not only do they have some additional (statistically oriented) methods

```
s.describe()
```

```
count    4.000000
mean    -0.013484
std      0.667092
min     -0.836113
25%     -0.408094
50%      0.082425
```

```
75%      0.477035
max      0.617328
Name: daily returns, dtype: float64
```

But their indices are more flexible

```
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—in this case, floats)

In fact, you can use much of the same syntax as Python dictionaries

```
s['AMZN']
```

```
0.43027108469945924
```

```
s['AMZN'] = 0
s
```

```
AMZN    0.000000
AAPL    0.617328
MSFT   -0.265421
GOOG   -0.836113
Name: daily returns, dtype: float64
```

```
'AAPL' in s
```

```
True
```

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, or as a structure for storing the $\mathbf{X}$ matrix in a linear regression

Let's look at an example that reads data from the CSV file `pandas/data/test_pwt.csv`, and can be downloaded here

Here's the contents of `test_pwt.csv`

```
"country","country isocode","year","POP","XRAT","tcgdp","cc","cg"
"Argentina","ARG","2000","37335.653","0.9995","295072.21869","75.716805379",
↪"5.5788042896"
"Australia","AUS","2000","19053.186","1.72483","541804.6521","67.759025993",
↪"6.7200975332"
"India","IND","2000","1006300.297","44.9416","1728144.3748","64.575551328",
↪"14.072205773"
"Israel","ISR","2000","6114.57","4.07733","129253.89423","64.436450847","10.
↪266688415"
"Malawi","MWI","2000","11801.505","59.543808333","5026.2217836","74.707624181
↪","11.658954494"
"South Africa","ZAF","2000","45064.098","6.93983","227242.36949","72.718710427
↪","5.7265463933"
"United States","USA","2000","282171.957","1","9898700","72.347054303","6.
↪0324539789"
"Uruguay","URY","2000","3219.793","12.099591667","25255.961693","78.978740282
↪","5.108067988"
```

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:

```
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
```

```
         country country isocode  year          POP       XRAT          ␣
↪tcgdp  cc         cg
0      Argentina             ARG  2000    37335.653   0.999500   295072.
↪218690   0  75.716805   5.578804
1      Australia             AUS  2000    19053.186   1.724830   541804.
↪652100   1  67.759026   6.720098
2          India             IND  2000  1006300.297  44.941600  1728144.
↪374800   2  64.575551  14.072206
3         Israel             ISR  2000     6114.570   4.077330   129253.
↪894230   3  64.436451  10.266688
4         Malawi             MWI  2000    11801.505  59.543808     5026.
↪221784   4  74.707624  11.658954
5   South Africa             ZAF  2000    45064.098   6.939830   227242.
↪369490   5  72.718710   5.726546
6  United States             USA  2000   282171.957   1.000000  9898700.
↪000000   6  72.347054   6.032454
7        Uruguay             URY  2000     3219.793  12.099592    25255.
↪961693   7  78.978740   5.108068
```

We can select particular rows using standard Python array slicing notation

```
df[2:5]
```

```
  country country isocode  year          POP       XRAT           tcgdp       ␣
↪  cc   cg
2   India             IND  2000  1006300.297  44.941600  1728144.374800  64.
↪575551   14.072206
3  Israel             ISR  2000     6114.570   4.077330   129253.894230  64.
↪436451   10.266688
4  Malawi             MWI  2000    11801.505  59.543808     5026.221784  74.
↪707624   11.658954
```

To select columns, we can pass a list containing the names of the desired columns represented as strings

```
df[['country', 'tcgdp']]
```

```
         country           tcgdp
0      Argentina   295072.218690
1      Australia   541804.652100
2          India  1728144.374800
3         Israel   129253.894230
4         Malawi     5026.221784
5   South Africa   227242.369490
6  United States  9898700.000000
7        Uruguay    25255.961693
```

To select both rows and columns using integers, the `iloc` attribute should be used with the format `.iloc[rows,columns]`

```
df.iloc[2:5,0:4]
```

```
  country country isocode  year          POP
2   India             IND  2000  1006300.297
3  Israel             ISR  2000     6114.570
4  Malawi             MWI  2000    11801.505
```

To select rows and columns using a mixture of integers and labels, the `loc` attribute can be used in a similar way

```
df.loc[df.index[2:5], ['country', 'tcgdp']]
```

```
  country           tcgdp
2   India  1728144.374800
3  Israel   129253.894230
4  Malawi     5026.221784
```

Let's imagine that we're 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
```

```
         country          POP           tcgdp
0      Argentina    37335.653   295072.218690
1      Australia    19053.186   541804.652100
2          India  1006300.297  1728144.374800
3         Israel     6114.570   129253.894230
4         Malawi    11801.505     5026.221784
5   South Africa    45064.098   227242.369490
6  United States   282171.957  9898700.000000
7        Uruguay     3219.793    25255.961693
```

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

```
df = df.set_index('country')
df
```

```
                       POP           tcgdp
country
Argentina        37335.653   295072.218690
Australia        19053.186   541804.652100
India          1006300.297  1728144.374800
Israel            6114.570   129253.894230
Malawi           11801.505     5026.221784
South Africa     45064.098   227242.369490
United States   282171.957  9898700.000000
Uruguay           3219.793    25255.961693
```

Let's give the columns slightly better names

```
df.columns = 'population', 'total GDP'
df
```

```
                population       total GDP
country
Argentina        37335.653   295072.218690
Australia        19053.186   541804.652100
India          1006300.297  1728144.374800
Israel            6114.570   129253.894230
Malawi           11801.505     5026.221784
South Africa     45064.098   227242.369490
United States   282171.957  9898700.000000
Uruguay           3219.793    25255.961693
```

Population is in thousands, let's revert to single units

```
df['population'] = df['population'] * 1e3
df
```

```
                population       total GDP
country
Argentina        37335653    295072.218690
Australia        19053186    541804.652100
India          1006300297   1728144.374800
Israel            6114570    129253.894230
Malawi           11801505      5026.221784
South Africa     45064098    227242.369490
United States   282171957   9898700.000000
Uruguay           3219793     25255.961693
```

Next we're going to add a column showing real GDP per capita, multiplying by 1,000,000 as we go because total GDP is in millions

```
df['GDP percap'] = df['total GDP'] * 1e6 / df['population']
df
```

```
               population       total GDP    GDP percap
country
Argentina        37335653   295072.218690   7903.229085
Australia        19053186   541804.652100  28436.433261
India          1006300297  1728144.374800   1717.324719
Israel            6114570   129253.894230  21138.672749
Malawi           11801505     5026.221784    425.896679
South Africa     45064098   227242.369490   5042.647686
United States   282171957  9898700.000000  35080.381854
Uruguay           3219793    25255.961693   7843.970620
```

One of the nice things about pandas `DataFrame` and `Series` objects is that they have methods for plotting and visualization that work through Matplotlib

For example, we can easily generate a bar plot of GDP per capita

```
df['GDP percap'].plot(kind='bar')
```

```
<matplotlib.axes.AxesSubplot at 0x2f22ed0>
```

```
import matplotlib.pyplot as plt

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

```
df = df.sort_values(by='GDP percap', ascending=False)
df
```

```
               population       total GDP    GDP percap
country
United States   282171957  9898700.000000  35080.381854
Australia        19053186   541804.652100  28436.433261
Israel            6114570   129253.894230  21138.672749
Argentina        37335653   295072.218690   7903.229085
Uruguay           3219793    25255.961693   7843.970620
South Africa     45064098   227242.369490   5042.647686
India          1006300297  1728144.374800   1717.324719
Malawi           11801505     5026.221784    425.896679
```

Plotting as before now yields

On-Line Data Sources

Python makes it straightforward to query on line databases programmatically

One important one 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

```
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')`

```
url = 'http://research.stlouisfed.org/fred2/series/UNRATE/downloaddata/UNRATE.
↪csv'
source = requests.get(url).content.decode().split("\n")
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 recognises our dates column, allowing for simple date filtering

```
data = pd.read_csv(url, index_col=0, parse_dates=True)
```

The data has been read into a pandas DataFrame called `data` that we can now manipulate in the usual way

```
type(data)
```

```
pandas.core.frame.DataFrame
```

```
data.head()  # A useful method to get a quick look at a data frame
```

```
            VALUE
DATE
1948-01-01    3.4
1948-02-01    3.8
1948-03-01    4.0
1948-04-01    3.9
1948-05-01    3.5
```

```
pd.set_option('precision', 1)
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

```
import matplotlib.pyplot as plt
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 France, Germany, the US and Australia

```
# == Get data and read into file gd.xls == #
wb_data_query = "http://api.worldbank.org/v2/en/indicator/gc.dod.totl.gd.zs?
↪downloadformat=excel"
r = requests.get(wb_data_query)
with open('gd.xls', 'wb') as output:
    output.write(r.content)

# == 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', 'DEU', 'FRA', '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

(Missing line segments indicate missing data values)

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',
               'KO': 'Coca-Cola',
```

```
                'GOOG': 'Google',
                'SNE': 'Sony',
                'PTR': 'PetroChina'}
```

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

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',
               'AAPL': 'Apple',
```

```
                'AMZN': 'Amazon',
                'BA': 'Boeing',
                'QCOM': 'Qualcomm',
                'KO': 'Coca-Cola',
                'GOOG': 'Google',
                'SNE': 'Sony',
                'PTR': 'PetroChina'}

price_change = {}

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

pc = pd.Series(price_change)
pc.sort_values(inplace=True)
fig, ax = plt.subplots(figsize=(10,8))
pc.plot(kind='bar', ax=ax)
plt.show()
```

Pandas for Panel Data

Contents

- *Pandas for Panel Data*
  - *Overview*
  - *Slicing and reshaping data*
  - *Merging dataframes and filling NaNs*
  - *Grouping and summarising data*
  - *Final Remarks*

– *Exercises*
– *Solutions*

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 summarising 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 summarised with the `groupby` function

Most of this lecture was created by Natasha Watkins

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

```
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')
```

Let's have a look at what we've got to work with

```
realwage.head()  # Show first 5 rows
```

	Unnamed: 0	Time	Country	Series	Pay period	value
0	0	2006-01-01	Ireland	In 2015 constant prices at 2015 USD PPPs	Annual	17,132.44
1	1	2007-01-01	Ireland	In 2015 constant prices at 2015 USD PPPs	Annual	18,100.92
2	2	2008-01-01	Ireland	In 2015 constant prices at 2015 USD PPPs	Annual	17,747.41
3	3	2009-01-01	Ireland	In 2015 constant prices at 2015 USD PPPs	Annual	18,580.14
4	4	2010-01-01	Ireland	In 2015 constant prices at 2015 USD PPPs	Annual	18,755.83

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

```
realwage = realwage.pivot_table(values='value',
                                index='Time',
                                columns=['Country', 'Series', 'Pay period'])
realwage.head()
```

Country			Ireland	...			Costa Rica
Series	In 2015 constant prices at 2015 USD PPPs		In 2015 constant prices at 2015 USD exchange rates	...	In 2015 constant prices at 2015 USD PPPs	In 2015 constant prices at 2015 USD exchange rates	
Pay period	Annual	Hourly	Annual	...	Hourly	Annual	Hourly
Time							
2006-01-01	17,132.44	8.24	19,090.54	...	nan	nan	nan
2007-01-01	18,100.92	8.70	20,169.70	...	nan	nan	nan
2008-01-01	17,747.41	8.53	19,775.78	...	nan	nan	nan
2009-01-01	18,580.14	8.93	20,703.69	...	nan	nan	nan
2010-01-01	18,755.83	9.02	20,899.47	...	nan	nan	nan

5 rows × 128 columns

To more easily filter our time series data later on, we will convert the index into a `DateTimeIndex`

```
realwage.index = pd.to_datetime(realwage.index)
type(realwage.index)
```

```
pandas.core.indexes.datetimes.DatetimeIndex
```

The columns contain multiple levels of indexing, known as a `MultiIndex`, with levels being ordered hierarchially (Country > Series > Pay period)

A `MultiIndex` is the simplest and most flexible way to manage panel data in pandas

```
type(realwage.columns)
```

```
pandas.core.indexes.multi.MultiIndex
```

```
realwage.columns.names
```

```
FrozenList(['Country', 'Series', 'Pay period'])
```

Like before, we can select the country (the top level of our `MultiIndex`)

```
realwage['United States'].head()
```

Series	In 2015 constant prices at 2015 USD PPPs		In 2015 constant prices at 2015 USD exchange rates	
Pay period	Annual	Hourly	Annual	Hourly
Time				
2006-01-01	12,594.40	6.05	12,594.40	6.05
2007-01-01	12,974.40	6.24	12,974.40	6.24
2008-01-01	14,097.56	6.78	14,097.56	6.78
2009-01-01	15,756.42	7.58	15,756.42	7.58
2010-01-01	16,391.31	7.88	16,391.31	7.88

Stacking and unstacking levels of the `MultiIndex` will be used throughout this lecture to reshape our dataframe into a format we need

`.stack()` rotates the lowest level of the column `MultiIndex` to the row index (`.unstack()` works in the opposite direction - try it out)

```
realwage.stack().head()
```

	Country		Australia	Belgium	...	United Kingdom		United States
	Series	In 2015 constant prices at 2015 USD PPPs	In 2015 constant prices at 2015 USD exchange rates	In 2015 constant prices at 2015 USD PPPs	...	In 2015 constant prices at 2015 USD exchange rates	In 2015 constant prices at 2015 USD PPPs	In 2015 constant prices at 2015 USD exchange rates
Time	Pay period							
2006-01-01	Annual	20,410.65	23,826.64	21,042.28	...	20,376.32	12,594.40	12,594.40
	Hourly	10.33	12.06	10.09	...	9.81	6.05	6.05
2007-01-01	Annual	21,087.57	24,616.84	21,310.05	...	20,954.13	12,974.40	12,974.40
	Hourly	10.67	12.46	10.22	...	10.07	6.24	6.24
2008-01-01	Annual	20,718.24	24,185.70	21,416.96	...	20,902.87	14,097.56	14,097.56

5 rows × 64 columns

We can also pass in an argument to select the level we would like to stack

```
realwage.stack(level='Country').head()
```

	Series	In 2015 constant prices at 2015 USD PPPs		In 2015 constant prices at 2015 USD exchange rates	
	Pay period	Annual	Hourly	Annual	Hourly
Time	Country				
2006-01-01	Australia	20,410.65	10.33	23,826.64	12.06
	Belgium	21,042.28	10.09	20,228.74	9.70
	Brazil	3,310.51	1.41	2,032.87	0.87
	Canada	13,649.69	6.56	14,335.12	6.89
	Chile	5,201.65	2.22	3,333.76	1.42

Using a `DatetimeIndex` makes it easy to select a particular time period

Selecting one year and stacking the two lower levels of the `MultiIndex` creates a cross-section of our panel data

```
realwage['2015'].stack(level=(1, 2)).transpose().head()
```

Time			2015-01-01	
Series	In 2015 constant prices at 2015 USD PPPs		In 2015 constant prices at 2015 USD exchange rates	
Pay period	Annual	Hourly	Annual	Hourly
Country				
Australia	21,715.53	10.99	25,349.90	12.83
Belgium	21,588.12	10.35	20,753.48	9.95
Brazil	4,628.63	2.00	2,842.28	1.21
Canada	16,536.83	7.95	17,367.24	8.35
Chile	6,633.56	2.80	4,251.49	1.81

For the rest of lecture, we will work with a dataframe of the hourly real minimum wages across countries and time, measured in 2015 US dollars

To create our filtered dataframe (`realwage_f`), we can use the `xs` method to select values at lower levels in the multiindex, while keeping the higher levels (countries in this case)

```
realwage_f = realwage.xs(('Hourly', 'In 2015 constant prices at 2015 USD␣
↪exchange rates'),
                         level=('Pay period', 'Series'), axis=1)
realwage_f.head()
```

Country Time	Australia	Belgium	Brazil	...	Turkey	United Kingdom	United States
2006-01-01	12.06	9.70	0.87	...	2.27	9.81	6.05
2007-01-01	12.46	9.82	0.92	...	2.26	10.07	6.24
2008-01-01	12.24	9.87	0.96	...	2.22	10.04	6.78
2009-01-01	12.40	10.21	1.03	...	2.28	10.15	7.58
2010-01-01	12.34	10.05	1.08	...	2.30	9.96	7.88

5 rows × 32 columns

Merging dataframes and filling NaNs

Similar to relational databases like SQL, pandas has built in methods to merge datasets together

Using country information from WorldData.info, we'll add the continent of each country to `realwage_f` with the `merge` function

The csv file can be found in `pandas_panel/countries.csv`, and can be downloaded here

```
worlddata = pd.read_csv('https://github.com/QuantEcon/QuantEcon.lectures.code/
↪raw/master/pandas_panel/countries.csv', sep=';')
worlddata.head()
```

	Country (en)	Country (de)	Country (local)	...	Deathrate	Life expectancy	Url
0	Afghanistan	Afghanistan	Afganistan/Afqanestan	...	13.70	51.30	https://www.laenderdaten.info/Asien/Afghanista...
1	Egypt	Ägypten	Misr	...	4.70	72.70	https://www.laenderdaten.info/Afrika/Aegypten/...
2	Åland Islands	Ålandinseln	Åland	...	0.00	0.00	https://www.laenderdaten.info/Europa/Aland/ind...
3	Albania	Albanien	Shqipëria	...	6.70	78.30	https://www.laenderdaten.info/Europa/Albanien/...
4	Algeria	Algerien	Al-Jaza'ir/Algérie	...	4.30	76.80	https://www.laenderdaten.info/Afrika/Algerien/...

5 rows × 17 columns

First we'll select just the country and continent variables from `worlddata` and rename the column to 'Country'

```
worlddata = worlddata[['Country (en)', 'Continent']]
worlddata = worlddata.rename(columns={'Country (en)': 'Country'})
worlddata.head()
```

	Country	Continent
0	Afghanistan	Asia
1	Egypt	Africa
2	Åland Islands	Europe
3	Albania	Europe
4	Algeria	Africa

We want to merge our new dataframe, `worlddata`, with `realwage_f`

The pandas `merge` function allows dataframes to be joined together by rows

Our dataframes will be merged using country names, requiring us to use the transpose of `realwage_f` so that rows correspond to country names in both dataframes

```
realwage_f.transpose().head()
```

Time	2006-01-01 00:00:00	2007-01-01 00:00:00	2008-01-01 00:00:00	...	2014-01-01 00:00:00	2015-01-01 00:00:00	2016-01-01 00:00:00
Country							
Australia	12.06	12.46	12.24	...	12.67	12.83	12.98
Belgium	9.70	9.82	9.87	...	10.01	9.95	9.76
Brazil	0.87	0.92	0.96	...	1.21	1.21	1.24
Canada	6.89	6.96	7.24	...	8.22	8.35	8.48
Chile	1.42	1.45	1.44	...	1.76	1.81	1.91

5 rows × 11 columns

We can use either left, right, inner, or outer join to merge our datasets:

- left join includes only countries from the left dataset
- right join includes only countries from the right dataset
- outer join includes countries that are in either the left and right datasets
- inner join includes only countries common to both the left and right datasets

By default, `merge` will use an inner join

Here we will pass `how='left'` to keep all countries in `realwage_f`, but discard countries in `worlddata` that do not have a corresponding data entry `realwage_f`

This is illustrated by the red shading in the following diagram

We will also need to specify where the country name is located in each dataframe, which will be the `key` that is used to merge the dataframes 'on'

Our 'left' dataframe (`realwage_f.transpose()`) contains countries in the index, so we set `left_index=True`

Our 'right' dataframe (`worlddata`) contains countries in the 'Country' column, so we set `right_on='Country'`

```
merged = pd.merge(realwage_f.transpose(), worlddata,
                  how='left', left_index=True, right_on='Country')
merged.head()
```

	2006-01-01 00:00:00	2007-01-01 00:00:00	2008-01-01 00:00:00	...	2016-01-01 00:00:00	Country	Continent
17	12.06	12.46	12.24	...	12.98	Australia	Australia
23	9.70	9.82	9.87	...	9.76	Belgium	Europe
32	0.87	0.92	0.96	...	1.24	Brazil	South America
100	6.89	6.96	7.24	...	8.48	Canada	North America
38	1.42	1.45	1.44	...	1.91	Chile	South America

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 occured, we can use `.isnull()` on the continent column and filter the merged dataframe

```
merged[merged['Continent'].isnull()]
```

	2006-01-01 00:00:00	2007-01-01 00:00:00	2008-01-01 00:00:00	...	2016-01-01 00:00:00	Country	Continent
247	3.42	3.74	3.87	...	5.28	Korea	NaN
247	0.23	0.45	0.39	...	0.55	Russian Federation	NaN
247	1.50	1.64	1.71	...	2.08	Slovak Republic	NaN

3 rows × 13 columns

We have three missing values!

One option to deal with NaN values is to create a dictionary containing these countries and their respective continents

`.map()` will match countries in `merged['Country']` with their continent from the dictionary

Notice how countries not in our dictionary are mapped with `NaN`

```
missing_continents = {'Korea': 'Asia',
                      'Russian Federation': 'Europe',
                      'Slovak Republic': 'Europe'}

merged['Country'].map(missing_continents)
```

```
17        NaN
23        NaN
32        NaN
100       NaN
38        NaN
108       NaN
41        NaN
225       NaN
53        NaN
58        NaN
45        NaN
68        NaN
233       NaN
86        NaN
88        NaN
91        NaN
247      Asia
117       NaN
122       NaN
123       NaN
138       NaN
153       NaN
151       NaN
174       NaN
175       NaN
247    Europe
247    Europe
```

```
198       NaN
200       NaN
227       NaN
241       NaN
240       NaN
Name: Country, dtype: object
```

We don't want to overwrite the entire series with this mapping

`.fillna()` only fills in `NaN` values in `merged['Continent']` with the mapping, while leaving other values in the column unchanged

```
merged['Continent'] = merged['Continent'].fillna(merged['Country'].
↪map(missing_continents))

# Check for whether continents were correctly mapped

merged[merged['Country'] == 'Korea']
```

	2006-01-01 00:00:00	2007-01-01 00:00:00	2008-01-01 00:00:00	...	2016-01-01 00:00:00	Country	Continent
247	3.42	3.74	3.87	...	5.28	Korea	Asia

1 rows × 13 columns

We will also combine the Americas into a single continent - this will make our visualisation nicer later on

To do this, we will use `.replace()` and loop through a list of the continent values we want to replace

```
replace = ['Central America', 'North America', 'South America']

for country in replace:
    merged['Continent'].replace(to_replace=country,
                                value='America',
                                inplace=True)
```

Now that we have all the data we want in a single `DataFrame`, we will reshape it back into panel form with a `MultiIndex`

We should also ensure to sort the index using `.sort_index()` so that we can efficiently filter our dataframe later on

By default, levels will be sorted top-down

```
merged = merged.set_index(['Continent', 'Country']).sort_index()
merged.head()
```

		2006-01-01 00:00:00	2007-01-01 00:00:00	2008-01-01 00:00:00	...	2014-01-01 00:00:00	2015-01-01 00:00:00	2016-01-01 00:00:00
Continent	**Country**							
America	**Brazil**	0.87	0.92	0.96	...	1.21	1.21	1.24
	Canada	6.89	6.96	7.24	...	8.22	8.35	8.48
	Chile	1.42	1.45	1.44	...	1.76	1.81	1.91
	Colombia	1.01	1.02	1.01	...	1.13	1.13	1.12
	Costa Rica	nan	nan	nan	...	2.41	2.56	2.63

5 rows × 11 columns

While merging, we lost our `DatetimeIndex`, as we merged columns that were not in datetime format

```
merged.columns
```

```
Index([2006-01-01 00:00:00, 2007-01-01 00:00:00, 2008-01-01 00:00:00,
       2009-01-01 00:00:00, 2010-01-01 00:00:00, 2011-01-01 00:00:00,
       2012-01-01 00:00:00, 2013-01-01 00:00:00, 2014-01-01 00:00:00,
       2015-01-01 00:00:00, 2016-01-01 00:00:00],
      dtype='object')
```

Now that we have set the merged columns as the index, we can recreate a `DatetimeIndex` using `.to_datetime()`

```
merged.columns = pd.to_datetime(merged.columns)
merged.columns = merged.columns.rename('Time')
merged.columns
```

```
DatetimeIndex(['2006-01-01', '2007-01-01', '2008-01-01', '2009-01-01',
               '2010-01-01', '2011-01-01', '2012-01-01', '2013-01-01',
               '2014-01-01', '2015-01-01', '2016-01-01'],
              dtype='datetime64[ns]', name='Time', freq=None)
```

The `DatetimeIndex` tends to work more smoothly in the row axis, so we will go ahead and transpose `merged`

```
merged = merged.transpose()
merged.head()
```

Continent		America		...			Europe
Country	Brazil	Canada	Chile	...	Slovenia	Spain	United Kingdom
Time							
2006-01-01	0.87	6.89	1.42	...	3.92	3.99	9.81
2007-01-01	0.92	6.96	1.45	...	3.88	4.10	10.07
2008-01-01	0.96	7.24	1.44	...	3.96	4.14	10.04
2009-01-01	1.03	7.67	1.52	...	4.08	4.32	10.15
2010-01-01	1.08	7.94	1.56	...	4.81	4.30	9.96

5 rows × 32 columns

Grouping and summarising data

Grouping and summarising data can be particularly useful for understanding large panel datasets

A simple way to summarise data is to call an aggregation method on the dataframe, such as `.mean()` or `.max()`

For example, we can calculate the average real minimum wage for each country over the period 2006 to 2016 (the default is to aggregate over rows)

```
merged.mean().head(10)
```

```
Continent  Country
America    Brazil          1.09
           Canada          7.82
           Chile           1.62
           Colombia        1.07
           Costa Rica      2.53
           Mexico          0.53
           United States   7.15
Asia       Israel          5.95
           Japan           6.18
           Korea           4.22
dtype: float64
```

Using this series, we can plot the average real minimum wage over the past decade for each country in our data set

```
import matplotlib.pyplot as plt
import matplotlib
matplotlib.style.use('seaborn')
```

```
%matplotlib inline

merged.mean().sort_values(ascending=False).plot(kind='bar', title="Average␣
↪real minimum wage 2006 - 2016")

#Set country labels
country_labels = merged.mean().sort_values(ascending=False).index.get_level_
↪values('Country').tolist()
plt.xticks(range(0, len(country_labels)), country_labels)
plt.xlabel('Country')

plt.show()
```

Passing in `axis=1` to `.mean()` will aggregate over columns (giving the average minimum wage for all countries over time)

```
merged.mean(axis=1).head()
```

```
Time
2006-01-01   4.69
2007-01-01   4.84
2008-01-01   4.90
2009-01-01   5.08
2010-01-01   5.11
dtype: float64
```

We can plot this time series as a line graph

```
merged.mean(axis=1).plot()
plt.title('Average real minimum wage 2006 - 2016')
plt.ylabel('2015 USD')
plt.xlabel('Year')
plt.show()
```

We can also specify a level of the `MultiIndex` (in the column axis) to aggregate over

```
merged.mean(level='Continent', axis=1).head()
```

Continent Time	America	Asia	Australia	Europe
2006-01-01	2.80	4.29	10.25	4.80
2007-01-01	2.85	4.44	10.73	4.94
2008-01-01	2.99	4.45	10.76	4.99
2009-01-01	3.23	4.53	10.97	5.16
2010-01-01	3.34	4.53	10.95	5.17

We can plot the average minimum wages in each continent as a time series

```
merged.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

```
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

```
merged.stack().describe()
```

Continent	America	Asia	Europe
count	69.00	44.00	200.00
mean	3.19	4.70	5.15
std	3.02	1.56	3.82
min	0.52	2.22	0.23
25%	1.03	3.37	2.02
50%	1.44	5.48	3.54
75%	6.96	5.95	9.70
max	8.48	6.65	12.39

This is a simplified way to use `groupby`

Using `groupby` generally follows a 'split-apply-combine' process:

- split: data is grouped based on one or more keys
- apply: a function is called on each group independently
- combine: the results of the function calls are combined into a new data structure

The `groupby` method achieves the first step of this process, creating a new `DataFrameGroupBy` object with data split into groups

Let's split `merged` by continent again, this time using the `groupby` function, and name the resulting object `grouped`

```
grouped = merged.groupby(level='Continent', axis=1)
grouped
```

```
<pandas.core.groupby.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`

```
grouped.size()
```

```
Continent
America     7
Asia        4
```

```
Europe     19
dtype: int64
```

Calling `.get_group()` to return just the countries in a single group, we can create a kernel density estimate of the distribution of real minimum wages in 2016 for each continent

`grouped.groups.keys()` will return the keys from the `groupby` object

```
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()
```

Final Remarks

This lecture has provided an introduction to some of pandas' more advanced features, including multi-indices, 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

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

Solutions

Exercise 1

```
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␣
↪format
employ.head()
```

UNIT	Percentage of total population			...			Thousand persons
AGE			From 15 to 24 years	...			From 55 to 64 years
SEX			Females	...			Total
INDIC_EM			Active population	...	Total employment (resident population concept - LFS)		
GEO	Austria	Belgium	Bulgaria	...	Switzerland	Turkey	United Kingdom
DATE							
2007-01-01	56.00	31.60	26.00	...	nan	1,282.00	4,131.00
2008-01-01	56.20	30.80	26.10	...	nan	1,354.00	4,204.00
2009-01-01	56.20	29.90	24.80	...	nan	1,449.00	4,193.00
2010-01-01	54.00	29.80	26.60	...	640.00	1,583.00	4,186.00
2011-01-01	54.80	29.80	24.80	...	661.00	1,760.00	4,164.00

5 rows × 1440 columns

This is a large dataset so it is useful to explore the levels and variables available

```
employ.columns.names
```

```
FrozenList(['UNIT', 'AGE', 'SEX', 'INDIC_EM', 'GEO'])
```

Variables within levels can be quickly retrieved with a loop

```
for name in employ.columns.names:
    print(name, employ.columns.get_level_values(name).unique())
```

```
UNIT Index(['Percentage of total population', 'Thousand persons'], dtype=
↪'object', name='UNIT')
AGE Index(['From 15 to 24 years', 'From 25 to 54 years', 'From 55 to 64 years
↪'], dtype='object', name='AGE')
SEX Index(['Females', 'Males', 'Total'], dtype='object', name='SEX')
INDIC_EM Index(['Active population', 'Total employment (resident population␣
↪concept - LFS)'], dtype='object', name='INDIC_EM')
GEO Index(['Austria', 'Belgium', 'Bulgaria', 'Croatia', 'Cyprus', 'Czech␣
↪Republic',
       'Denmark', 'Estonia', 'Euro area (17 countries)',
       'Euro area (18 countries)', 'Euro area (19 countries)',
       'European Union (15 countries)', 'European Union (27 countries)',
       'European Union (28 countries)', 'Finland',
       'Former Yugoslav Republic of Macedonia, the', 'France',
       'France (metropolitan)',
       'Germany (until 1990 former territory of the FRG)', 'Greece', 'Hungary
↪',
       'Iceland', 'Ireland', 'Italy', 'Latvia', 'Lithuania', 'Luxembourg',
       'Malta', 'Netherlands', 'Norway', 'Poland', 'Portugal', 'Romania',
       'Slovakia', 'Slovenia', 'Spain', 'Sweden', 'Switzerland', 'Turkey',
```

```
       'United Kingdom'],
      dtype='object', name='GEO')
```

Exercise 2

To easily filter by country, swap `GEO` to the top level and sort the `MultiIndex`

```
employ.columns = employ.columns.swaplevel(0,-1)
employ = employ.sort_index(axis=1)
```

We need to get rid of a few items in `GEO` which are not countries

A fast way to get rid of the EU areas is to use a list comprehension to find the level values in `GEO` that begin with 'Euro'

```
geo_list = employ.columns.get_level_values('GEO').unique().tolist()
countries = [x for x in geo_list if not x.startswith('Euro')]
employ = employ[countries]
employ.columns.get_level_values('GEO').unique()
```

```
Index(['Austria', 'Belgium', 'Bulgaria', 'Croatia', 'Cyprus', 'Czech Republic
↪',
       'Denmark', 'Estonia', 'Finland',
       'Former Yugoslav Republic of Macedonia, the', 'France',
       'France (metropolitan)',
       'Germany (until 1990 former territory of the FRG)', 'Greece', 'Hungary
↪',
       'Iceland', 'Ireland', 'Italy', 'Latvia', 'Lithuania', 'Luxembourg',
       'Malta', 'Netherlands', 'Norway', 'Poland', 'Portugal', 'Romania',
       'Slovakia', 'Slovenia', 'Spain', 'Sweden', 'Switzerland', 'Turkey',
       'United Kingdom'],
      dtype='object', name='GEO')
```

Select only percentage employed in the active population from the dataframe

```
employ_f = employ.xs(('Percentage of total population', 'Active population'),
                     level=('UNIT', 'INDIC_EM'),
                     axis=1)
employ_f.head()
```

GEO	Austria			...	United Kingdom		
AGE	From 15 to 24 years			...	From 55 to 64 years		
SEX	Females	Males	Total	...	Females	Males	Total
DATE							
2007-01-01	56.00	62.90	59.40	...	49.90	68.90	59.30
2008-01-01	56.20	62.90	59.50	...	50.20	69.80	59.80
2009-01-01	56.20	62.90	59.50	...	50.60	70.30	60.30
2010-01-01	54.00	62.60	58.30	...	51.10	69.20	60.00
2011-01-01	54.80	63.60	59.20	...	51.30	68.40	59.70

5 rows × 306 columns

Drop the 'Total' value before creating the grouped boxplot

```
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()
```

Linear Regression in Python

Contents

- *Linear Regression in Python*
  - *Overview*
  - *Simple Linear Regression*
  - *Extending the Linear Regression Model*
  - *Endogeneity*
  - *Summary*
  - *Exercises*
  - *Solutions*

Overview

Linear regression is a standard tool for analyzing the relationship between two or more variables

In this lecture we'll use the Python package `statsmodels` to estimate, interpret, and visualize linear regression models

Along the way we'll discuss a variety of topics, including

- simple and multivariate linear regression
- visualization
- endogeneity and omitted variable bias
- two-stage least squares

As an example, we will replicate results from Acemoglu, Johnson and Robinson's seminal paper *[AJR01]*

- You can download a copy here

In the paper, the authors emphasize the importance of institutions in economic development

The main contribution is the use of settler mortality rates as a source of *exogenous* variation in institutional differences

Such variation is needed to determine whether it is institutions that give rise to greater economic growth, rather than the other way around

Prerequisites

This lecture assumes you are familiar with basic econometrics

For an introductory text covering these topics, see, for example, *[Woo15]*

Comments

Most of the credit for this lecture goes to Natasha Watkins

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 Serivces Group

These variables and other data used in the paper are available for download on Daron Acemoglu's webpage

We will use pandas' `.read_stata()` function to read in data contained in the `.dta` files to dataframes

```
import pandas as pd

df1 = pd.read_stata('https://github.com/QuantEcon/QuantEcon.lectures.code/raw/
↪master/ols/maketable1.dta')
df1.head()
```

	shortnam	euro1900	excolony	avexpr	logpgp95	cons1	cons90	democ00a	cons00a	extmort4	logem4	loghjypl	baseco
0	AFG	0.000000	1.0	NaN	NaN	1.0	2.0	1.0	1.0	93.699997	4.540098	NaN	NaN
1	AGO	8.000000	1.0	5.363636	7.770645	3.0	3.0	0.0	1.0	280.000000	5.634789	-3.411248	1.0
2	ARE	0.000000	1.0	7.181818	9.804219	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	ARG	60.000004	1.0	6.386364	9.133459	1.0	6.0	3.0	3.0	68.900002	4.232656	-0.872274	1.0
4	ARM	0.000000	0.0	NaN	7.682482	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

```
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:

- β_0 is the intercept of the linear trend line on the y-axis
- β_1 is the slope of the linear trend line, representing the *marginal effect* of protection against risk on log GDP per capita
- u_i is a random error term (deviations of observations from the linear trend due to factors not included in the model)

Visually, this linear model involves choosing a straight line that best fits the data, as in the following plot (Figure 2 in *[AJR01]*)

```
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 (β's) of the linear model is Ordinary Least Squares (OLS)

As the name implies, an OLS model is solved by finding the parameters that minimize *the sum of squared residuals*, ie.

$$\min_{\hat{\beta}} \sum_{i=1}^{N} \hat{u}_i^2$$

where $\hat{u}_i$ is the difference between the observation and the predicted value of the dependent variable

To estimate the constant term β_0, we need to add a column of 1's to our dataset (consider the equation if β_0 was replaced with $\beta_0 x_i$ and $x_i = 1$)

```
df1['const'] = 1
```

Now we can construct our model in `statsmodels` using the OLS function

We will use `pandas` dataframes with `statsmodels`, however standard arrays can also be used as arguments

```
import statsmodels.api as sm
```

```
reg1 = sm.OLS(endog=df1['logpgp95'], exog=df1[['const', 'avexpr']], missing=
↪'drop')
type(reg1)
```

```
statsmodels.regression.linear_model.OLS
```

So far we have simply constructed our model

We need to use `.fit()` to obtain parameter estimates $\hat{\beta}_0$ and $\hat{\beta}_1$

```
results = reg1.fit()
type(results)
```

```
statsmodels.regression.linear_model.RegressionResultsWrapper
```

We now have the fitted regression model stored in `results`

To view the OLS regression results, we can call the `.summary()` method

Note that an observation was mistakenly dropped from the results in the original paper (see the note located in *maketable2.do* from Acemoglu's webpage), and thus the coefficients differ slightly

```
print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:               logpgp95   R-squared:                       0.611
Model:                            OLS   Adj. R-squared:                  0.608
Method:                 Least Squares   F-statistic:                     171.4
Date:                Mon, 17 Jul 2017   Prob (F-statistic):           4.16e-24
Time:                        18:41:28   Log-Likelihood:                -119.71
No. Observations:                 111   AIC:                             243.4
Df Residuals:                     109   BIC:                             248.8
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          4.6261      0.301     15.391      0.000       4.030       5.222
avexpr         0.5319      0.041     13.093      0.000       0.451       0.612
==============================================================================
Omnibus:                        9.251   Durbin-Watson:                   1.689
Prob(Omnibus):                  0.010   Jarque-Bera (JB):                9.170
Skew:                          -0.680   Prob(JB):                       0.0102
Kurtosis:                       3.362   Cond. No.                         33.2
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is␣
↪correctly specified.
```

From our results, we see that

- The intercept $\hat{\beta}_0 = 4.63$
- The slope $\hat{\beta}_1 = 0.53$
- The positive $\hat{\beta}_1$ parameter estimate implies that institutional quality has a positive effect on economic outcomes, as we saw in the figure
- The p-value of 0.000 for $\hat{\beta}_1$ implies that the effect of institutions on GDP is statistically significant (using p < 0.05 as a rejection rule)
- The R-squared value of 0.611 indicates that around 61% of variation in log GDP per capita is explained by protection against expropriation

Using our parameter estimates, we can now write our estimated relationship as

$$\widehat{logpgp95}_i = 4.63 + 0.53 \; avexpr_i$$

This equation describes the line that best fits our data, as shown in Figure 2

We can use this equation to predict the level of log GDP per capita for a value of the index of expropriation protection

For example, for a country with an index value of 7.07 (the average for the dataset), we find that their predicted level of log GDP per capita in 1995 is 8.38

```
mean_expr = np.mean(df1_subset['avexpr'])
mean_expr
```

```
6.515625
```

```
predicted_logpdp95 = 4.63 + 0.53 * 7.07
predicted_logpdp95
```

```
8.3771
```

An easier (and more accurate) way to obtain this result is to use `.predict()` and set $constant = 1$ and $avexpr_i = mean_expr$

```
results.predict(exog=[1, mean_expr])
```

```
array([ 8.09156367])
```

We can obtain an array of predicted $logpgp95_i$ for every value of $avexpr_i$ in our dataset by calling `.predict()` on our results

Plotting the predicted values against $avexpr_i$ shows that the predicted values lie along the linear line that we fitted above

The observed values of $logpgp95_i$ are also plotted for comparison purposes

```
# Drop missing observations from whole sample

df1_plot = df1.dropna(subset=['logpgp95', 'avexpr'])
```

```
# Plot predicted values

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()
```

Extending the Linear Regression Model

So far we have only accounted for institutions affecting economic performance - almost certainly there are numerous other factors affecting GDP that are not included in our model

Leaving out variables that affect $logpgp95_i$ will result in **omitted variable bias**, yielding biased and inconsistent parameter estimates

We can extend our bivariate regression model to a **multivariate regression model** by adding in other factors that may affect $logpgp95_i$

[AJR01] consider other factors such as:

- the effect of climate on economic outcomes; latitude is used to proxy this
- differences that affect both economic performance and institutions, eg. cultural, historical, etc.; controlled for with the use of continent dummies

Let's estimate some of the extended models considered in the paper (Table 2) using data from `maketable2.dta`

```
df2 = pd.read_stata('https://github.com/QuantEcon/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()
```

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)

```
from statsmodels.iolib.summary2 import summary_col

info_dict={'R-squared' : lambda x: "{:.2f}".format(x.rsquared),
           'No. observations' : lambda x: "{0:d}".format(int(x.nobs))}

results_table = summary_col(results=[reg1,reg2,reg3],
                            float_format='%0.2f',
                            stars = True,
                            model_names=['Model 1',
                                         'Model 3',
                                         'Model 4'],
                            info_dict=info_dict,
                            regressor_order=['const',
                                             'avexpr',
                                             'lat_abst',
                                             'asia',
                                             'africa'])

results_table.add_title('Table 2 - OLS Regressions')
```

```
print(results_table)
```

```
        Table 2 - OLS Regressions
=========================================
                 Model 1 Model 3 Model 4
-----------------------------------------
const            4.63*** 4.87*** 5.85***
                 (0.30)  (0.33)  (0.34)
avexpr           0.53*** 0.46*** 0.39***
                 (0.04)  (0.06)  (0.05)
lat_abst                 0.87*   0.33
                         (0.49)  (0.45)
asia                             -0.15
                                 (0.15)
africa                           -0.92***
                                 (0.17)
other                            0.30
                                 (0.37)
R-squared        0.61    0.62    0.72
No. observations 111     111     111
=========================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
```

Endogeneity

As *[AJR01]* discuss, the OLS models likely suffer from **endogeneity** issues, resulting in biased and inconsistent model estimates

Namely, there is likely a two-way relationship between institutions and economic outcomes:

- richer countries may be able to afford or prefer better institutions
- variables that affect income may also be correlated with institutional differences
- the construction of the index may be biased; analysts may be biased towards seeing countries with higher income having better institutions

To deal with endogeneity, we can use **two-stage least squares (2SLS) regression**, which is an extension of OLS regression

This method requires replacing the endogenous variable $avexpr_i$ with a variable that is:

1. correlated with $avexpr_i$
2. not correlated with the error term (ie. it should not directly affect the dependent variable, otherwise it would be correlated with u_i due to omitted variable bias)

The new set of regressors is called an **instrument**, which aims to remove endogeneity in our proxy of institutional differences

The main contribution of *[AJR01]* is the use of settler mortality rates to instrument for institutional differences

They hypothesize that higher mortality rates of colonizers led to the establishment of institutions that were more extractive in nature (less protection against expropriation), and these institutions still persist today

Using a scatterplot (Figure 3 in *[AJR01]*), we can see protection against expropriation is negatively correlated with settler mortality rates, coinciding with the authors' hypothesis and satisfying the first condition of a valid instrument

```
# Dropping NA's is required to use numpy's polyfit
df1_subset2 = df1.dropna(subset=['logem4', 'avexpr'])

X = df1_subset2['logem4']
y = df1_subset2['avexpr']
labels = df1_subset2['shortnam']

# Replace markers with country labels
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 satisifed 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 $logem4_i$

Therefore, we will estimate the first-stage regression as

$$avexpr_i = \delta_0 + \delta_1 logem4_i + v_i$$

The data we need to estimate this equation is located in `maketable4.dta` (only complete data, indicated by `baseco = 1`, is used for estimation)

```
# Import and select the data
df4 = pd.read_stata('https://github.com/QuantEcon/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())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 avexpr   R-squared:                       0.270
Model:                            OLS   Adj. R-squared:                  0.258
Method:                 Least Squares   F-statistic:                     22.95
Date:                Mon, 17 Jul 2017   Prob (F-statistic):           1.08e-05
Time:                        18:41:29   Log-Likelihood:                -104.83
No. Observations:                  64   AIC:                             213.7
Df Residuals:                      62   BIC:                             218.0
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          9.3414      0.611     15.296      0.000       8.121      10.562
logem4        -0.6068      0.127     -4.790      0.000      -0.860      -0.354
==============================================================================
Omnibus:                        0.035   Durbin-Watson:                   2.003
Prob(Omnibus):                  0.983   Jarque-Bera (JB):                0.172
Skew:                           0.045   Prob(JB):                        0.918
Kurtosis:                       2.763   Cond. No.                         19.4
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is␣
↪correctly specified.
```

Second stage

We need to retrieve the predicted values of $avexpr_i$ using `.predict()`

We then replace the endogenous variable $avexpr_i$ with the predicted values $\widehat{avexpr}_i$ in the original linear model

Our second stage regression is thus

$$logpgp95_i = \beta_0 + \beta_1 \widehat{avexpr}_i + u_i$$

```
df4['predicted_avexpr'] = results_fs.predict()

results_ss = sm.OLS(df4['logpgp95'],
                    df4[['const', 'predicted_avexpr']]).fit()
print(results_ss.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:               logpgp95   R-squared:                       0.477
Model:                            OLS   Adj. R-squared:                  0.469
Method:                 Least Squares   F-statistic:                     56.60
Date:                Mon, 17 Jul 2017   Prob (F-statistic):           2.66e-10
Time:                        18:41:29   Log-Likelihood:                -72.268
No. Observations:                  64   AIC:                             148.5
Df Residuals:                      62   BIC:                             152.9
Df Model:                           1
Covariance Type:            nonrobust
==================================================================================
                      coef    std err          t      P>|t|      [0.025      ␣
↪0.975]
----------------------------------------------------------------------------
↪------
const               1.9097      0.823      2.320      0.024       0.264      ␣
↪ 3.555
predicted_avexpr    0.9443      0.126      7.523      0.000       0.693      ␣
↪ 1.195
==============================================================================
Omnibus:                       10.547   Durbin-Watson:                   2.137
Prob(Omnibus):                  0.005   Jarque-Bera (JB):               11.010
Skew:                          -0.790   Prob(JB):                      0.00407
Kurtosis:                       4.277   Cond. No.                         58.1
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is␣
↪correctly specified.
```

The second-stage regression results give us an unbiased and consistent estimate of the effect of institutions on economic outcomes

The result suggests a stronger positive relationship than what the OLS results indicated

Note that while our parameter estimates are correct, our standard errors are not and for this reason, computing 2SLS 'manually' (in stages with OLS) is not recommended

We can correctly estimate a 2SLS regression in one step using the linearmodels package, an extension of `statsmodels`

To install this package, you will need to run `pip install linearmodels` in your command line

```
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)

```
iv = IV2SLS(dependent=df4['logpgp95'],
            exog=df4['const'],
            endog=df4['avexpr'],
            instruments=df4['logem4']).fit(cov_type='unadjusted')

print(iv.summary)
```

```
                          IV-2SLS Estimation Summary
==============================================================================
Dep. Variable:                logpgp95   R-squared:                      0.1870
Estimator:                     IV-2SLS   Adj. R-squared:                 0.1739
No. Observations:                   64   F-statistic:                    37.568
Date:                 Mon, Jul 17 2017   P-value (F-stat)                0.0000
Time:                         18:41:29   Distribution:                  chi2(1)
Cov. Estimator:             unadjusted

                             Parameter Estimates
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
const          1.9097     1.0106     1.8897     0.0588     -0.0710      3.8903
avexpr         0.9443     0.1541     6.1293     0.0000      0.6423      1.2462
==============================================================================

Endogenous: avexpr
Instruments: logem4
Unadjusted Covariance (Homoskedastic)
Debiased: False
```

Given that we now have consistent and unbiased estimates, we can infer from the model we have estimated that institutional differences (stemming from institutions set up during colonization) can explain 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

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

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

$$H_0 : Cov(avexpr_i, u_i) = 0 \quad (no\ endogeneity)$$
$$H_1 : Cov(avexpr_i, u_i) \neq 0 \quad (endogeneity)$$

This test is run is two stages

First, we regress $avexpr_i$ on the instrument, $logem4_i$

$$avexpr_i = \pi_0 + \pi_1 logem4_i + \upsilon_i$$

Second, we retrieve the residuals $\hat{\upsilon}_i$ and include them in the original equation

$$logpgp95_i = \beta_0 + \beta_1 avexpr_i + \alpha\hat{\upsilon}_i + u_i$$

If α is statistically significant (with a p-value < 0.05), then we reject the null hypothesis and conclude that $avexpr_i$ is endogenous

Using the above information, estimate a Hausman test and interpret your results

Exercise 2

The OLS parameter β can also be estimated using matrix algebra and `numpy` (you may need to review the numpy lecture to complete this exercise)

The linear equation we want to estimate is (written in matrix form)

$$y = X\beta + u$$

To solve for the unknown parameter β, we want to minimise the sum of squared residuals

$$\min_{\hat{\beta}} \hat{u}'\hat{u}$$

Rearranging the first equation and substituting into the second equation, we can write

$$\min_{\hat{\beta}} (Y - X\hat{\beta})'(Y - X\hat{\beta})$$

Solving this optimisation 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

Solutions

Exercise 1

```
# 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
==============================================================================
Dep. Variable:               logpgp95   R-squared:                       0.689
Model:                            OLS   Adj. R-squared:                  0.679
Method:                 Least Squares   F-statistic:                     74.05
Date:                Mon, 17 Jul 2017   Prob (F-statistic):           1.07e-17
Time:                        18:41:29   Log-Likelihood:                -62.031
No. Observations:                  70   AIC:                             130.1
Df Residuals:                      67   BIC:                             136.8
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          2.4782      0.547      4.530      0.000       1.386       3.570
avexpr         0.8564      0.082     10.406      0.000       0.692       1.021
resid         -0.4951      0.099     -5.017      0.000      -0.692      -0.298
==============================================================================
Omnibus:                       17.597   Durbin-Watson:                   2.086
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               23.194
Skew:                          -1.054   Prob(JB):                     9.19e-06
Kurtosis:                       4.873   Cond. No.                         53.8
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is␣
↪correctly specified.
```

The output shows that the coefficient on the residuals is statistically significant, indicating $avexpr_i$ is endogenous

Exercise 2

```
# 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 beta hat
beta_hat = np.linalg.solve(X.T @ X, X.T @ y)

# Print out the results from the 2 x 1 vector beta_hat
print('beta0: ', np.round(beta_hat, 2)[0])
print('beta1: ', np.round(beta_hat, 2)[1])
```

```
beta0:  4.63
beta1:  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

CHAPTER

FOUR

PART 4: TOOLS AND TECHNIQUES

This section of the course contains foundational mathematical and statistical tools and techniques

Linear Algebra

Contents

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

Overview

Linear algebra is one of the most useful branches of applied mathematics for economists to invest in

For example, many applied problems in economics and finance require the solution of a linear system of equations, such as

$$\begin{aligned} y_1 &= ax_1 + bx_2 \\ y_2 &= cx_1 + dx_2 \end{aligned}$$

or, more generally,

$$\begin{aligned} y_1 &= a_{11}x_1 + a_{12}x_2 + \cdots + a_{1k}x_k \\ &\vdots \\ y_n &= a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nk}x_k \end{aligned} \tag{4.1}$$

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

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

If you're interested, the Python code for producing this figure is here

Vector Operations

The two most common operators for vectors are addition and scalar multiplication, which we now describe

As a matter of definition, when we add two vectors, we add them element by element

$$x + y = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} := \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix}$$

Scalar multiplication is an operation that takes a number γ and a vector x and produces

$$\gamma x := \begin{bmatrix} \gamma x_1 \\ \gamma x_2 \\ \vdots \\ \gamma x_n \end{bmatrix}$$

Scalar multiplication is illustrated in the next figure

In 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

```
import numpy as np

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

```
np.sum(x * y)           # Inner product of x and y
```

```
12.0
```

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

```
1.7320508075688772
```

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

```
1.7320508075688772
```

Span

Given a set of vectors $A := \{a_1, \ldots, a_k\}$ in $\mathbb{R}^n$, it's natural to think about the new vectors we can create by performing linear operations

New vectors created in this manner are called *linear combinations* of A

In particular, $y \in \mathbb{R}^n$ is a linear combination of $A := \{a_1, \ldots, a_k\}$ if

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

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

The code for producing this figure can be found here

Examples

If A contains only one vector $a_1 \in \mathbb{R}^2$, then its span is just the scalar multiples of a_1, which is the unique line passing through both a_1 and the origin

If $A = \{e_1, e_2, e_3\}$ consists of the *canonical basis vectors* of $\mathbb{R}^3$, that is

$$e_1 := \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad e_2 := \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad e_3 := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

then the span of A is all of $\mathbb{R}^3$, because, for any $x = (x_1, x_2, x_3) \in \mathbb{R}^3$, we can write

$$x = x_1 e_1 + x_2 e_2 + x_3 e_3$$

Now consider $A_0 = \{e_1, e_2, e_1 + e_2\}$

If $y = (y_1, y_2, y_3)$ is any linear combination of these vectors, then $y_3 = 0$ (check it)

Hence A_0 fails to span all of $\mathbb{R}^3$

Linear Independence

As we'll see, it's often desirable to find families of vectors with relatively large span, so that many vectors can be described by linear operators on a few vectors

The condition we need for a set of vectors to have a large span is what's called linear independence

In particular, a collection of vectors $A := \{a_1, \ldots, a_k\}$ 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

Matrices

Matrices are a neat way of organizing data for use in linear operations

An $n \times k$ matrix is a rectangular array A of numbers with n rows and k columns:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}$$

Often, the numbers in the matrix represent coefficients in a system of linear equations, as discussed at the start of this lecture

For obvious reasons, the matrix A is also called a vector if either $n = 1$ or $k = 1$

In the former case, A is called a *row vector*, while in the latter it is called a *column vector*

If $n = k$, then A is called *square*

The matrix formed by replacing a_{ij} by a_{ji} for every i and j is called the *transpose* of A, and denoted A' or $A^\top$

If $A = A'$, then A is called *symmetric*

For a square matrix A, the i elements of the form a_{ii} for $i = 1, \ldots, n$ are called the *principal diagonal*

A is called *diagonal* if the only nonzero entries are on the principal diagonal

If, in addition to being diagonal, each element along the principal diagonal is equal to 1, then A is called the *identity matrix*, and denoted by I

Matrix Operations

Just as was the case for vectors, a number of algebraic operations are defined for matrices

Scalar multiplication and addition are immediate generalizations of the vector case:

$$\gamma A = \gamma \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} := \begin{bmatrix} \gamma a_{11} & \cdots & \gamma a_{1k} \\ \vdots & \vdots & \vdots \\ \gamma a_{n1} & \cdots & \gamma a_{nk} \end{bmatrix}$$

and

$$A + B = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} + \begin{bmatrix} b_{11} & \cdots & b_{1k} \\ \vdots & \vdots & \vdots \\ b_{n1} & \cdots & b_{nk} \end{bmatrix} := \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1k} + b_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} + b_{n1} & \cdots & a_{nk} + b_{nk} \end{bmatrix}$$

In the latter case, the matrices must have the same shape in order for the definition to make sense

We also have a convention for *multiplying* two matrices

The rule for matrix multiplication generalizes the idea of inner products discussed above, and is designed to make multiplication play well with basic linear operations

If A and B are two matrices, then their product AB is formed by taking as its i, j-th element the inner product of the i-th row of A and the j-th column of B

There are many tutorials to help you visualize this operation, such as this one, or the discussion on the Wikipedia page

If A is $n \times k$ and B is $j \times m$, then to multiply A and B we require $k = j$, and the resulting matrix AB is $n \times m$

As perhaps the most important special case, consider multiplying $n \times k$ matrix A and $k \times 1$ column vector x

According to the preceding rule, this gives us an $n \times 1$ column vector

$$Ax = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} := \begin{bmatrix} a_{11}x_1 + \cdots + a_{1k}x_k \\ \vdots \\ a_{n1}x_1 + \cdots + a_{nk}x_k \end{bmatrix} \tag{4.2}$$

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

```
import numpy as np

A = ((1, 2),
     (3, 4))

type(A)
```

```
tuple
```

```
A = np.array(A)

type(A)
```

[1] Although there is a specialized matrix data type defined in NumPy, it's more standard to work with ordinary NumPy arrays. See *this discussion*.

```
numpy.ndarray
```

```
A.shape
```

```
(2, 2)
```

The `shape` attribute is a tuple giving the number of rows and columns — see *here* for more discussion

To get the transpose of `A`, use `A.transpose()` or, more simply, `A.T`

There are many convenient functions for creating common matrices (matrices of zeros, ones, etc.) — see *here*

Since operations are performed elementwise by default, scalar multiplication and addition have very natural syntax

```
A = np.identity(3)
B = np.ones((3, 3))
2 * A
```

```
array([[ 2.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0.,  2.]])
```

```
A + B
```

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

To multiply matrices we use `np.dot`

In particular, `np.dot(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 $f \colon \mathbb{R}^k \to \mathbb{R}^n$ is called *linear* if, for all $x, y \in \mathbb{R}^k$ and all scalars α, β, we have

$$f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$$

You can check that this holds for the function $f(x) = Ax + b$ when b is the zero vector, and fails when b is nonzero

In fact, it's known that f is linear if and *only if* there exists a matrix A such that $f(x) = Ax$ for all x.

Solving Systems of Equations

Recall again the system of equations (4.1)

If we compare (4.1) and (4.2), we see that (4.1) can now be written more conveniently as

$$y = Ax \tag{4.3}$$

The problem we face is to determine a vector $x \in \mathbb{R}^k$ that solves (4.3), taking y and A as given

This is a special case of a more general problem: Find an x such that $y = f(x)$

Given an arbitrary function f and a y, is there always an x such that $y = f(x)$?

If so, is it always unique?

The answer to both these questions is negative, as the next figure shows

In 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 (4.3) 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

$$Ax = x_1 a_1 + \cdots + x_k a_k$$

Hence the range of $f(x) = Ax$ is exactly the span of the columns of A

We want the range to be large, so that it contains arbitrary y

As you might recall, the condition that we want for the span to be large is *linear independence*

A happy fact is that linear independence of the columns of A also gives us uniqueness

Indeed, it follows from our *earlier discussion* that if $\{a_1, \ldots, a_k\}$ are linearly independent and $y = Ax = x_1 a_1 + \cdots + x_k a_k$, then no $z \neq x$ satisfies $y = Az$

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 we're looking for

Determinants

Another quick comment about square matrices is that to every such matrix we assign a unique number called the *determinant* of the matrix — you can find the expression for it here

If the determinant of A is not zero, then we say that A is *nonsingular*

Perhaps the most important fact about determinants is that A is nonsingular if and only if A is of full column rank

This gives us a useful one-number summary of whether or not a square matrix can be inverted

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 $\mathbb{R}^2$

This set can never be linearly independent, since it is possible to find two vectors that span $\mathbb{R}^2$

(For example, use the canonical basis vectors)

It follows that one column is a linear combination of the other two

For example, let's say that $a_1 = \alpha a_2 + \beta a_3$

Then if $y = Ax = x_1 a_1 + x_2 a_2 + x_3 a_3$, we can also write

$$y = x_1(\alpha a_2 + \beta a_3) + x_2 a_2 + x_3 a_3 = (x_1 \alpha + x_2) a_2 + (x_1 \beta + x_3) a_3$$

In other words, uniqueness fails

Linear Equations with SciPy

Here's an illustration of how to solve linear equations with SciPy's `linalg` submodule

All of these routines are Python front ends to time-tested and highly optimized FORTRAN code

```
import numpy as np
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 = np.dot(A_inv, y)  # Solution
np.dot(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 `np.dot(inv(A), y)`, or using `solve(A, y)`

The latter method uses a different algorithm (LU decomposition) that is numerically more stable, and hence should almost always be preferred

To obtain the least squares solution $\hat{x} = (A'A)^{-1}A'y$, use `scipy.linalg.lstsq(A, y)`

Eigenvalues and Eigenvectors

Let A be an $n \times n$ square matrix

If λ is scalar and v is a non-zero vector in $\mathbb{R}^n$ such that

$$Av = \lambda v$$

then we say that λ is an *eigenvalue* of A, and v is an *eigenvector*

Thus, an eigenvector of A is a vector such that when the map $f(x) = Ax$ is applied, v is merely scaled

The next figure shows two eigenvectors (blue arrows) and their images under A (red arrows)

As expected, the image Av of each v is just a scaled version of the original

The eigenvalue equation is equivalent to $(A - \lambda I)v = 0$, and this has a nonzero solution v only when the columns of $A - \lambda I$ are linearly dependent

This in turn is equivalent to stating that the determinant is zero

Hence to find all eigenvalues, we can look for λ such that the determinant of $A - \lambda I$ is zero

This problem can be expressed as one of solving for the roots of a polynomial in λ of degree n

This in turn implies the existence of n solutions in the complex plane, although some might be repeated

Some nice facts about the eigenvalues of a square matrix A are as follows

1. The determinant of A equals the product of the eigenvalues
2. The trace of A (the sum of the elements on the principal diagonal) equals the sum of the eigenvalues
3. If A is symmetric, then all of its eigenvalues are real
4. If A is invertible and $\lambda_1, \ldots, \lambda_n$ 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

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

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 λ and eigenvectors v such that

$$Av = \lambda Bv$$

This can be solved in SciPy via `scipy.linalg.eig(A, B)`

Of course, if B is square and invertible, then we can treat the generalized eigenvalue problem as an ordinary eigenvalue problem $B^{-1}Av = \lambda v$, but this is not always the case

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[2]

Neumann's Theorem

Let A be a square matrix and let $A^k := AA^{k-1}$ with $A^1 := A$

In other words, A^k is the k-th power of A

Neumann's theorem states the following: If $\|A^k\| < 1$ for some $k \in \mathbb{N}$, then $I - A$ is invertible, and

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k \tag{4.4}$$

Spectral Radius

A result known as Gelfand's formula tells us that, for any square matrix A,

$$\rho(A) = \lim_{k \to \infty} \|A^k\|^{1/k}$$

Here $\rho(A)$ is the *spectral radius*, defined as $\max_i |\lambda_i|$, where $\{\lambda_i\}_i$ is the set of eigenvalues of A

As a consequence of Gelfand's formula, if all eigenvalues are strictly less than one in modulus, there exists a k with $\|A^k\| < 1$

In which case (4.4) is valid

[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.

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 $x \in \mathbb{R}^n$

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 $n \times 1$ vectors
- A be an $n \times n$ matrix
- B be an $m \times n$ matrix and y be an $m \times 1$ vector

Then

1. $\frac{\partial a'x}{\partial x} = a$
2. $\frac{\partial Ax}{\partial x} = A'$
3. $\frac{\partial x'Ax}{\partial x} = (A + A')x$
4. $\frac{\partial y'Bz}{\partial y} = Bz$
5. $\frac{\partial y'Bz}{\partial B} = yz'$

Exercise 1 below asks you to apply these formulas

Further Reading

The documentation of the `scipy.linalg` submodule can be found here

Chapters 2 and 3 of the Econometric Theory contains a discussion of linear algebra along the same lines as above, with solved exercises

If you don't mind a slightly abstract approach, a nice intermediate-level text on linear algebra is *[Janich94]*

Exercises

Exercise 1

Let x be a given $n \times 1$ vector and consider the problem

$$v(x) = \max_{y,u} \{-y'Py - u'Qu\}$$

subject to the linear constraint

$$y = Ax + Bu$$

Here

- P is an $n \times n$ matrix and Q is an $m \times m$ matrix
- A is an $n \times n$ matrix and B is an $n \times m$ matrix
- both P and Q are symmetric and positive semidefinite

(What must the dimensions of y and u be to make this a well-posed problem?)

One way to solve the problem is to form the Lagrangian

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

where λ is an $n \times 1$ vector of Lagrange multipliers

Try applying the formulas given above for differentiating quadratic and linear forms to obtain the first-order conditions for maximizing $\mathcal{L}$ with respect to y, u and minimizing it with respect to λ

Show that these conditions imply that

1. $\lambda = -2Py$
2. The optimizing choice of u satisfies $u = -(Q + B'PB)^{-1}B'PAx$
3. The function v satisfies $v(x) = -x'\tilde{P}x$ where $\tilde{P} = A'PA - A'PB(Q + B'PB)^{-1}B'PA$

As we will see, in economic contexts Lagrange multipliers often are shadow prices

---

Note: If we don't care about the Lagrange multipliers, we can subsitute the constraint into the objective function, and then just maximize $-(Ax + Bu)'P(Ax + Bu) - u'Qu$ with respect to u. You can verify that this leads to the same maximizer.

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-oder 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-oder condition for maximizing L w.r.t. u

Thus, the optimial choice of u must satisfy

$$u = -(Q + B'PB)^{-1}B'PAx \,,$$

which follows from the definition of the first-oder 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 optimial choice of u satisfies $ u = -(Q + B'PB)^{-1}B'PAx $, then

$$v(x) = -(Ax + Bu)'P(Ax + Bu) - u'Qu \quad with \quad u = -(Q + B'PB)^{-1}B'PAx$$

To evaluate the function

$$\begin{aligned} v(x) &= -(Ax + Bu)'P(Ax + Bu) - u'Qu \\ &= -(x'A' + u'B')P(Ax + Bu) - u'Qu \\ &= -x'A'PAx - u'B'PAx - x'A'PBu - u'B'PBu - u'Qu \\ &= -x'A'PAx - 2u'B'PAx - u'(Q + B'PB)u \end{aligned}$$

For simplicity, denote by $S := (Q + B'PB)^{-1}B'PA$, then $ u = -Sx$

Regarding the second term $-2u'B'PAx$,

$$\begin{aligned} -2u'B'PAx &= -2x'S'B'PAx \\ &= 2x'A'PB(Q + B'PB)^{-1}B'PAx \end{aligned}$$

Notice that the term $(Q + B'PB)^{-1}$ is symmetic as both P and Q are symmetric

Regarding the third term $-u'(Q + B'PB)u$,

$$\begin{aligned} -u'(Q + B'PB)u &= -x'S'(Q + B'PB)Sx \\ &= -x'A'PB(Q + B'PB)^{-1}B'PAx \end{aligned}$$

Hence, the summation of second and third terms is $x'A'PB(Q + B'PB)^{-1}B'PAx$

This implies that

$$\begin{aligned} v(x) &= -x'A'PAx - 2u'B'PAx - u'(Q + B'PB)u \\ &= -x'A'PAx + x'A'PB(Q + B'PB)^{-1}B'PAx \\ &= -x'[A'PA - A'PB(Q + B'PB)^{-1}B'PA]x \end{aligned}$$

Therefore, the solution to the optimization problem $v(x) = -x'\tilde{P}x$ follows the above result by denoting $\tilde{P} := A'PA - A'PB(Q + B'PB)^{-1}B'PA$

Orthogonal Projections and Their Applications

Contents

- *Orthogonal Projections and Their Applications*

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

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

Key Definitions

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

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

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

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

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

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

The **orthogonal complement** of linear subspace $S \subset RR^n$ 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 we'll *see below*

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

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

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

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

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

The next figure provides some intuition

Proof of sufficiency

We'll omit the full proof.

But we will prove sufficiency of the asserted conditions

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

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

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

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

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

Orthogonal Projection as a Mapping

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

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

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

In what follows we denote this operator by a matrix P

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

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

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

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

From this we can deduce additional useful properties, such as

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

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

Orthogonal Complement

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

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

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

We write

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

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

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

This amounts to another version of the OPT:

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

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

The next figure illustrates

Orthonormal Basis

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

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

If O is orthonormal and $\operatorname{span} O = S$, then O is called an **orthonormal basis** of S

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

One example of an orthonormal set is the canonical basis $\{e_1, \ldots, e_n\}$ that forms an orthonormal basis of $\mathbb{R}^n$, where e_i is the :math:'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 $x \in \operatorname{span}\{u_1, \ldots, u_k\}$, we can find scalars $\alpha_1, \ldots, \alpha_k$ that verify

$$x = \sum_{j=1}^{k} \alpha_j u_j \tag{4.5}$$

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 (4.5) verifies the claim

Projection onto an Orthonormal Basis

When the subspace onto which are projecting is orthonormal, computing the projection simplifies:

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

$$Py = \sum_{i=1}^{k} \langle y, u_i \rangle u_i, \quad \forall \, y \in \mathbb{R}^n \tag{4.6}$$

Proof: Fix $y \in \mathbb{R}^n$ and let Py be defined as in (4.6)

Clearly, $Py \in S$

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

It sufficies to show that $y - Py \perp$ any basis vector u_i (why?)

This is true because

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

Projection Using Matrix Algebra

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

We want to compute the matrix P that verifies

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

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

This reference is useful <https://en.wikipedia.org/wiki/Linear_map#Matrices>‘_

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

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

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

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

Claim 1 is true because

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

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

Claim 2 is equivalent to the statement

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

This is true: If $b \in \mathbb{R}^K$, then

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

The proof is now complete

Starting with X

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

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

Then the columns of X form a basis of S

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

We know that the projection of y onto S is

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

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

Hence

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

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

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 $\operatorname{span}(X)$ we have

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

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

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

This is what we aimed to show

Least Squares Regression

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

This approach provides insights about many geometric properties of linear regression

We treat only some examples

Squared risk measures

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

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

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

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

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

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

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

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

If we let $\mathcal{F}$ be the class of linear functions $1/N$, the problem is

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

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

Solution

Define the matrices

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

and

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

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

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

Since monotone transforms don't affect minimizers, we have

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

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

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

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

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

The **vector of fitted values** is

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

The **vector of residuals** is

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

Here are some more standard definitions:

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

  TSS = ESS + SSR

We can prove this easily using the OPT

From the OPT we have $y = \hat{y} + \hat{u}$ and $\hat{u} \perp \hat{y}$

Applying the Pythagorean law completes the proof

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

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

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

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

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

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

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

- $x_j := \mathrm{col}_j(X)$
- $\{u_1, \ldots, u_k\}$ 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 $x_j \in \mathrm{span}\{u_1, \ldots, u_j\}$, 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

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

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

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?

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

```
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

    """

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

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

    for i in range(1, k):
        # Set up
        b = X[:,i]        # The vector we're going to project
        Z = X[:, 0:i]    # first i-1 columns of X

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

```
        u = M @ b

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

    return U
```

Here are the arrays we'll work with

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

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

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

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

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

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

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

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

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

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

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

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

```
from scipy.linalg import qr

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

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

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

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

Again, we obtain the same answer

Finite Markov Chains

Contents

- *Finite Markov Chains*
  - *Overview*
  - *Definitions*
  - *Simulation*
  - *Marginal Distributions*
  - *Irreducibility and Aperiodicity*
  - *Stationary Distributions*
  - *Ergodicity*
  - *Computing Expectations*
  - *Exercises*
  - *Solutions*

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

Prerequisite knowledge is basic probability and linear algebra

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 $y \in S$,

$$\mathbb{P}\{X_{t+1} = y \,|\, X_t\} = \mathbb{P}\{X_{t+1} = y \,|\, X_t, X_{t-1}, \ldots\} \tag{4.7}$$

In other words, knowing the current state is enough to know probabilities for future states

In particular, the dynamics of a Markov chain are fully determined by the set of values

$$P(x, y) := \mathbb{P}\{X_{t+1} = y \,|\, X_t = x\} \qquad (x, y \in S) \tag{4.8}$$

By construction,

- $P(x, y)$ is the probability of going from x to y in one unit of time (one step)
- $P(x, \cdot)$ is the conditional distribution of X_{t+1} given $X_t = x$

We can view P as a stochastic matrix where

$$P_{ij} = P(x_i, x_j) \qquad 1 \leq i, j \leq n$$

Going the other way, if we take a stochastic matrix P, we can generate a Markov chain $\{X_t\}$ as follows:

- 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 (4.8)

[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.

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 employed worker loses her job and becomes unemployed with probability $\beta \in (0, 1)$
2. An unemployed worker finds a job with probability $\alpha \in (0, 1)$

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 α and β, we can address a range of questions, such as

- What is the average duration of unemployment?
- Over the long-run, what fraction of time does a worker find herself unemployed?
- Conditional on employment, what is the probability of becoming unemployed at least once over the next 12 months?

We'll cover such applications below

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

Here "ng" is normal growth, "mr" is mild recession, etc.

Simulation

One natural way to answer questions about Markov chains is to simulate them

(To approximate the probability of event E, we can simulate many times and count the fraction of times that E occurs)

Nice functionality for simulating Markov chains exists in QuantEcon

- Efficient, bundled with lots of other useful routines for handling Markov chains

However, it's also a good exercise to roll our own routines — let's do that first and then come back to the methods in QuantEcon

In these exercises we'll 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 ψ 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 ψ
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

```
import quantecon as qe

psi = (0.1, 0.9)           # Probabilities over sample space {0, 1}
d = qe.DiscreteRV(psi)
d.draw(5)                  # Generate 5 independent draws from psi
```

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

We'll write our code as a function that takes the following three arguments

- A stochastic matrix `P`
- An initial state `init`
- A positive integer `sample_size` representing the length of the time series the function should return

```
import numpy as np

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 = [qe.DiscreteRV(P[i,:]) for i in range(n)]

    # === generate the sample path === #
    for t in range(sample_size - 1):
        X[t+1] = P_dist[X[t]].draw()

    return X
```

Let's see how it works using the small matrix

$$P := \begin{pmatrix} 0.4 & 0.6 \\ 0.2 & 0.8 \end{pmatrix} \tag{4.9}$$

As we'll see later, for a long series drawn from `P`, the fraction of the sample that takes value 0 will be about 0.25

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 QuantEcon's Routines

As discussed above, QuantEcon 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 routine is *JIT compiled* and much faster

(Because it's 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=('employed', 'unemployed'))
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])
```

Marginal Distributions

Suppose that

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

What then is the distribution of X_{t+1}, or, more generally, of X_{t+m}?

Solution

Let ψ_t be the distribution of X_t for $t = 0, 1, 2, \ldots$

Our first aim is to find ψ_{t+1} given ψ_t and P

To begin, pick any $y \in S$

Using the law of total probability, we can decompose the probability that $X_{t+1} = y$ as follows:

$$\mathbb{P}\{X_{t+1} = y\} = \sum_{x \in S} \mathbb{P}\{X_{t+1} = y \,|\, X_t = x\} \cdot \mathbb{P}\{X_t = x\}$$

In words, to get the probability of being at y tomorrow, we account for all ways this can happen and sum their probabilities

Rewriting this statement in terms of marginal and conditional probabilities gives

$$\psi_{t+1}(y) = \sum_{x \in S} P(x, y)\psi_t(x)$$

There are n such equations, one for each $y \in S$

If we think of ψ_{t+1} and ψ_t as *row vectors* (as is traditional in this literature), these n equations are summarized by the matrix expression

$$\psi_{t+1} = \psi_t P \tag{4.10}$$

In other words, to move the distribution forward one unit of time, we postmultiply by P

By repeating this m times we move forward m steps into the future

Hence, iterating on (4.10), the expression $\psi_{t+m} = \psi_t P^m$ is also valid — here P^m is the m-th power of P As a special case, we see that if ψ_0 is the initial distribution from which X_0 is drawn, then $\psi_0 P^m$ is the distribution of X_m

This is very important, so let's repeat it

$$X_0 \sim \psi_0 \quad \Longrightarrow \quad X_m \sim \psi_0 P^m \tag{4.11}$$

and, more generally,

$$X_t \sim \psi_t \quad \Longrightarrow \quad X_{t+m} \sim \psi_t P^m \tag{4.12}$$

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 (4.12), but now with ψ_t putting all probability on state x

- 1 in the x-th position and zero elsewhere

Inserting this into (4.12), we see that, conditional on $X_t = x$, the distribution of X_{t+m} is the x-th row of P^m

In particular

$$\mathbb{P}\{X_{t+m} = y\} = P^m(x, y) = (x, y)\text{-th element of } P^m$$

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 ψ 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

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 ficticious 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's MarkovChain class

```
P = [[0.9, 0.1, 0.0],
     [0.4, 0.4, 0.2],
     [0.1, 0.1, 0.8]]
```

```
mc = qe.MarkovChain(P, ('poor', 'middle', 'rich'))
mc.is_irreducible
```

```
True
```

Here's a more pessimistic scenario, where the poor are poor forever

This stochastic matrix is not irreducible, since, for example, *rich* is not accessible from *poor*

Let's confirm this

```
P = [[1.0, 0.0, 0.0],
     [0.1, 0.8, 0.1],
     [0.0, 0.2, 0.8]]

mc = qe.MarkovChain(P, ('poor', 'middle', 'rich'))
mc.is_irreducible
```

```
False
```

We can also determine the "communication classes"

```
mc.communication_classes
```

```
[array(['poor'], dtype='<U6'),
 array(['middle', 'rich'], dtype='<U6')]
```

It might be clear to you already that irreducibility is going to be important in terms of long run outcomes

For example, poverty is a life sentence in the second graph but not the first

We'll come back to this a bit later

Aperiodicity

Loosely speaking, a Markov chain is called periodic if it cycles in a predictible way, and aperiodic otherwise

Here's a trivial example with three states

The chain cycles with period 3:

```
P = [[0, 1, 0],
     [0, 0, 1],
     [1, 0, 0]]

mc = qe.MarkovChain(P)
mc.period
```

```
3
```

More formally, the **period** of a state x is the 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

We can confirm that the stochastic matrix is periodic as follows

```
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
```

Stationary Distributions

As seen in (4.10), we can shift probabilities forward one unit of time via postmultiplication by P

Some distributions are invariant under this updating process — for example,

```
P = np.array([[.4, .6], [.2, .8]])
psi = (0.25, 0.75)
np.dot(psi, 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 $\psi^* = \psi^* P$

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 — we'll discuss this more in just a moment

Mathematically, a stationary distribution is a fixed point of P when P is thought of as the map $\psi \mapsto \psi P$ from (row) vectors to (row) vectors

Theorem. Every stochastic matrix P has at least one stationary distribution

(We are assuming here that the state space S is finite; if not more assumptions are required)

For 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 ψ^*
2. For any initial distribution ψ_0, we have $\|\psi_0 P^t - \psi^*\| \to 0$ as $t \to \infty$

For a proof, see, for example, theorem 5.2 of *[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 ψ such that $\psi = \psi P$

In fact if P has two distinct stationary distributions ψ_1, ψ_2 then it has infinitely many, since in this case, as you can verify,

$$\psi_3 := \lambda \psi_1 + (1 - \lambda) \psi_2$$

is a stationary distribuiton for P for any $\lambda \in [0, 1]$

If we restrict attention to the case where only one stationary distribution exists, one option for finding it is to try to solve the linear system $\psi(I_n - P) = 0$ for ψ, where I_n is the $n \times n$ identity

But the zero vector solves this equation

Hence we need to impose the restriction that the solution must be a probability distribution

A suitable algorithm is implemented in QuantEcon — the next code block illustrates

```
P = [[0.4, 0.6], [0.2, 0.8]]
mc = qe.MarkovChain(P)
mc.stationary_distributions  # Show all stationary distributions
```

```
array([[ 0.25,  0.75]])
```

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 ψ^* as a stochastic steady state

The convergence in the theorem is illustrated in the next figure

Here

- P is the stochastic matrix for recession and growth *considered above*
- The highest red dot is an arbitrarily chosen initial probability distribution ψ, represented as a vector in $\mathbb{R}^3$
- The other red dots are the distributions ψP^t for $t = 1, 2, \ldots$
- The black dot is ψ^*

The code for the figure can be found here — you might like to try experimenting with different initial conditions

Ergodicity

Under irreducibility, yet another important result obtains: For all $x \in S$,

$$\frac{1}{n} \sum_{t=1}^{n} \mathbf{1}\{X_t = x\} \to \psi^*(x) \quad \text{as } n \to \infty \tag{4.13}$$

Here

- $\mathbf{1}\{X_t = x\} = 1$ if $X_t = x$ and zero otherwise
- convergence is with probability one
- the result does not depend on the distribution (or value) of X_0

The result tells us that the fraction of time the chain spends at state x converges to $\psi^*(x)$ as time goes to infinity This gives us another way to interpret the stationary distribution — provided that the convergence result in (4.13) is valid

The convergence in (4.13) 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

Computing Expectations

We are interested in computing expectations of the form

$$\mathbb{E}[h(X_t)] \tag{4.14}$$

and conditional expectations such as

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] \tag{4.15}$$

where

- $\{X_t\}$ is a Markov chain generated by $n \times n$ stochastic matrix P
- h is a given function, which, in expressions involving matrix algebra, we'll think of as the column vector

$$h = \begin{pmatrix} h(x_1) \\ \vdots \\ h(x_n) \end{pmatrix}$$

The unconditional expectation (4.14) is easy: We just sum over the distribution of X_t to get

$$\mathbb{E}[h(X_t)] = \sum_{x \in S} (\psi P^t)(x) h(x)$$

Here ψ is the distribution of X_0

Since ψ and hence ψP^t are row vectors, we can also write this as

$$\mathbb{E}[h(X_t)] = \psi P^t h$$

For the conditional expectation (4.15), we need to sum over the conditional distribution of X_{t+k} given $X_t = x$

We already know that this is $P^k(x, \cdot)$, so

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] = (P^k h)(x) \tag{4.16}$$

The vector $P^k h$ stores the conditional expectation $\mathbb{E}[h(X_{t+k}) \mid X_t = x]$ 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

$$\mathbb{E}\left[\sum_{j=0}^{\infty} \beta^j h(X_{t+j}) \mid X_t = x\right] = [(I - \beta P)^{-1} h](x)$$

where

$$(I - \beta P)^{-1} = I + \beta P + \beta^2 P^2 + \cdots$$

Premultiplication by $(I - \beta P)^{-1}$ amounts to "applying the **resolvent operator**"

Exercises

Exercise 1

According to the discussion *immediately above*, if a worker's employment dynamics obey the stochastic matrix

$$P = \begin{pmatrix} 1 - \alpha & \alpha \\ \beta & 1 - \beta \end{pmatrix}$$

with $\alpha \in (0,1)$ and $\beta \in (0,1)$, then, in the long-run, the fraction of time spent unemployed will be

$$p := \frac{\beta}{\alpha + \beta}$$

In other words, if $\{X_t\}$ represents the Markov chain for employment, then $\bar{X}_n \to p$ as $n \to \infty$, where

$$\bar{X}_n := \frac{1}{n} \sum_{t=1}^{n} \mathbf{1}\{X_t = 0\}$$

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 $\bar{X}_n - p$ against n, 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

(You don't need to add the fancy touches to the graph—see the solution if you're interested)

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

- ℓ_i is the total number of outbound links from i
- L_j is the set of all pages i such that i has a link to j

This is a measure of the number of inbound links, weighted by their own ranking (and normalized by $1/\ell_i$)

There is, however, another interpretation, and it brings us back to Markov chains

Let P be the matrix given by $P(i,j) = \mathbf{1}\{i \to j\}/\ell_i$ where $\mathbf{1}\{i \to j\} = 1$ if i has a link to j and zero otherwise

The matrix P is a stochastic matrix provided that each page has at least one link

With this definition of P we have

$$r_j = \sum_{i \in L_j} \frac{r_i}{\ell_i} = \sum_{\text{all } i} \mathbf{1}\{i \to j\} \frac{r_i}{\ell_i} = \sum_{\text{all } i} P(i,j) r_i$$

Writing r for the row vector of rankings, this becomes $r = rP$

Hence r is the stationary distribution of the stochastic matrix P

Let's think of $P(i,j)$ as the probability of "moving" from page i to page j

The value $P(i,j)$ has the interpretation

- $P(i,j) = 1/k$ if i has k outbound links, and j is one of them
- $P(i,j) = 0$ if i has no direct link to j

Thus, motion from page to page is that of a web surfer who moves from one page to another by randomly clicking on one of the links on that page

Here "random" means that each link is selected with equal probability

Since r is the stationary distribution of P, assuming that the uniform ergodicity condition is valid, we *can interpret* r_j as the fraction of time that a (very persistent) random surfer spends at page j

Your exercise is to apply this ranking algorithm to the graph pictured above, and return the list of pages ordered by rank

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

```
import re

re.findall('\w', 'x +++ y ****** z')  # \w matches alphanumerics
```

```
['x', 'y', 'z']
```

```
re.findall('\w', 'a ^^ b &&& $$ c')
```

```
['a', 'b', 'c']
```

When you solve for the ranking, you will find that the highest ranked node is in fact `g`, while the lowest is `a`

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 *[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 $N(0, \sigma_u^2)$

The values $P(x_i, x_j)$ are computed to approximate the AR(1) process — omitting the derivation, the rules are as follows:

1. If $j = 0$, then set

$$P(x_i, x_j) = P(x_i, x_0) = F(x_0 - \rho x_i + s/2)$$

2. If $j = n - 1$, then set

$$P(x_i, x_j) = P(x_i, x_{n-1}) = 1 - F(x_{n-1} - \rho x_i - s/2)$$

3. Otherwise, set

$$P(x_i, x_j) = F(x_j - \rho x_i + s/2) - F(x_j - \rho x_i - s/2)$$

The exercise is to write a function `approx_markov(rho, sigma_u, m=3, n=7)` that returns $\{x_0, \ldots, x_{n-1}\} \subset \mathbb{R}$ and $n \times n$ matrix P as described above

- Even better, write a function that returns an instance of QuantEcon.py's *MarkovChain* class

Solutions

```
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 = ((1 - alpha, alpha),    # Careful: P and p are distinct
     (beta, 1 - beta))
P = np.array(P)
mc = MarkovChain(P)

fig, ax = plt.subplots(figsize=(9, 6))
ax.set_ylim(-0.25, 0.25)
ax.grid()
ax.hlines(0, 0, N, lw=2, alpha=0.6)  # Horizonal line at zero

for x0, col in ((0, 'blue'), (1, 'green')):
    # == Generate time series for worker that starts at x0 == #
    X = mc.simulate(N, init=x0)
    # == Compute fraction of time spent unemployed, for each n == #
    X_bar = (X == 0).cumsum() / (1 + np.arange(N, dtype=float))
    # == Plot == #
```

```
    ax.fill_between(range(N), np.zeros(N), X_bar - p, color=col, alpha=0.1)
    ax.plot(X_bar - p, color=col, label=r'$X_0 = \, {} $'.format(x0))
    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

```
%%file web_graph_data.txt
a -> d;
a -> f;
b -> j;
b -> k;
b -> m;
c -> c;
c -> g;
c -> j;
c -> m;
d -> f;
d -> h;
d -> k;
```

```
e -> d;
e -> h;
e -> l;
f -> a;
f -> b;
f -> j;
f -> l;
g -> b;
g -> j;
h -> d;
h -> g;
h -> l;
h -> m;
i -> g;
i -> h;
i -> n;
j -> e;
j -> i;
j -> k;
k -> n;
l -> m;
m -> g;
n -> c;
n -> j;
n -> m;
```

```
Overwriting web_graph_data.txt
```

```
"""
Return list of pages, ordered by rank
"""
import numpy as np
from operator import itemgetter
import re

infile = 'web_graph_data.txt'
alphabet = 'abcdefghijklmnopqrstuvwxyz'

n = 14 # Total number of web pages (nodes)

# == Create a matrix Q indicating existence of links == #
#  * Q[i, j] = 1 if there is a link from i to j
#  * Q[i, j] = 0 otherwise
Q = np.zeros((n, n), dtype=int)
f = open(infile, 'r')
edges = f.readlines()
f.close()
for edge in edges:
    from_node, to_node = re.findall('\w', edge)
    i, j = alphabet.index(from_node), alphabet.index(to_node)
    Q[i, j] = 1
# == Create the corresponding Markov matrix P == #
```

```
P = np.empty((n, n))
for i in range(n):
    P[i,:] = Q[i,:] / Q[i,:].sum()
mc = MarkovChain(P
                 )
# == Compute the stationary distribution r == #
r = mc.stationary_distributions[0]
ranked_pages = {alphabet[i] : r[i] for i in range(n)}
# == Print solution, sorted from highest to lowest rank == #
print('Rankings\n ***')
for name, rank in sorted(ranked_pages.items(), key=itemgetter(1), reverse=1):
    print('{0}: {1:.4}'.format(name, rank))
```

```
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 library can be found here

LLN and CLT

Contents

- *LLN and CLT*
  - *Overview*
  - *Relationships*
  - *LLN*
  - *CLT*
  - *Exercises*

– *Solutions*

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

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

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 *Kolmogorov's strong law*

Let $X_1, \ldots, X_n$ be independent and identically distributed scalar random variables, with common distribution F

When it exists, let μ denote the common mean of this sample:

$$\mu := \mathbb{E}X = \int x F(dx)$$

In addition, let

$$\bar{X}_n := \frac{1}{n} \sum_{i=1}^{n} X_i$$

Kolmogorov's strong law states that, if $\mathbb{E}|X|$ is finite, then

$$\mathbb{P}\left\{ \bar{X}_n \to \mu \text{ as } n \to \infty \right\} = 1 \tag{4.17}$$

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, (4.17) 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 $\mathbb{E}X_i^2 < \infty$, then, for any $\epsilon > 0$, we have

$$\mathbb{P}\left\{ |\bar{X}_n - \mu| \geq \epsilon \right\} \to 0 \quad \text{as} \quad n \to \infty \tag{4.18}$$

(This version is weaker because we claim only convergence in probability rather than almost sure convergence, and assume a finite second moment)

To see that this is so, fix $\epsilon > 0$, and let σ^2 be the variance of each X_i

Recall the Chebyshev inequality, which tells us that

$$\mathbb{P}\left\{ |\bar{X}_n - \mu| \geq \epsilon \right\} \leq \frac{\mathbb{E}[(\bar{X}_n - \mu)^2]}{\epsilon^2} \tag{4.19}$$

Now observe that

$$\begin{aligned}
\mathbb{E}[(\bar{X}_n - \mu)^2] &= \mathbb{E}\left\{ \left[\frac{1}{n} \sum_{i=1}^{n} (X_i - \mu) \right]^2 \right\} \\
&= \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \mathbb{E}(X_i - \mu)(X_j - \mu) \\
&= \frac{1}{n^2} \sum_{i=1}^{n} \mathbb{E}(X_i - \mu)^2 \\
&= \frac{\sigma^2}{n}
\end{aligned}$$

Here the crucial step is at the third equality, which follows from independence

Independence means that if $i \neq j$, then the covariance term $\mathbb{E}(X_i - \mu)(X_j - \mu)$ drops out

As a result, $n^2 - n$ terms vanish, leading us to a final expression that goes to zero in n

Combining our last result with (4.19), we come to the estimate

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \leq \frac{\sigma^2}{n\epsilon^2} \tag{4.20}$$

The claim in (4.18) is now clear

Of course, if the sequence $X_1, \ldots, X_n$ is correlated, then the cross-product terms $\mathbb{E}(X_i - \mu)(X_j - \mu)$ are not necessarily zero

While this doesn't mean that the same line of argument is impossible, it does mean that if we want a similar result then the covariances should be "almost zero" for "most" of these terms

In a long sequence, this would be true if, for example, $\mathbb{E}(X_i - \mu)(X_j - \mu)$ approached zero when the difference between i and j became large

In other words, the LLN can still work if the sequence $X_1, \ldots, X_n$ has a kind of "asymptotic independence", in the sense that correlation falls to zero as variables become further apart in the sequence

This idea is very important in time series analysis, and we'll come across it again soon enough

Illustration

Let's now illustrate the classical IID law of large numbers using simulation

In particular, we aim to generate some sequences of IID random variables and plot the evolution of $\bar{X}_n$ as n increases

Below is a figure that does just this (as usual, you can click on it to expand it)

It shows IID observations from three different distributions and plots $\bar{X}_n$ against n in each case

The dots represent the underlying observations X_i for $i = 1, \ldots, 100$

In each of the three cases, convergence of $\bar{X}_n$ to μ occurs as predicted

The figure was produced by file `illustrates_lln.py`, and can be downloaded here

The three distributions are chosen at random from a selection stored in the dictionary `distributions`

```
"""
Filename: illustrates_lln.py
Authors: John Stachurski and Thomas J. Sargent

Visual illustration of the law of large numbers.
"""

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 == #
num_plots = 3
fig, axes = plt.subplots(num_plots, 1, figsize=(10, 10))

# == Set some plotting parameters to improve layout == #
bbox = (0., 1.02, 1., .102)
legend_args = {'ncol': 2,
               'bbox_to_anchor': bbox,
               'loc': 3,
               'mode': 'expand'}
plt.subplots_adjust(hspace=0.5)

for ax in axes:
    # == Choose a randomly selected distribution == #
    name = random.choice(list(distributions.keys()))
    distribution = distributions.pop(name)

    # == Generate n draws from the distribution == #
    data = distribution.rvs(n)

    # == Compute sample mean at each n == #
    sample_mean = np.empty(n)
    for i in range(n):
        sample_mean[i] = np.mean(data[:i+1])

    # == Plot == #
    ax.plot(list(range(n)), data, 'o', color='grey', alpha=0.5)
    axlabel = r'$\bar X_n$' + ' for ' + r'$X_i \sim$' + ' ' + name
    ax.plot(list(range(n)), sample_mean, 'g-', lw=3, alpha=0.6, label=axlabel)
    m = distribution.mean()
    ax.plot(list(range(n)), [m] * n, 'k--', lw=1.5, label=r'$\mu$')
    ax.vlines(list(range(n)), m, data, lw=0.2)
    ax.legend(**legend_args)

plt.show()
```

Infinite Mean

What happens if the condition $\mathbb{E}|X| < \infty$ 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

$$f(x) = \frac{1}{\pi(1+x^2)} \qquad (x \in \mathbb{R})$$

The next figure shows 100 independent draws from this distribution

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

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

$$\phi(t) = \mathbb{E}e^{itX} = \int e^{itx} f(x) dx = e^{-|t|} \tag{4.21}$$

Using independence, the characteristic function of the sample mean becomes

$$\begin{aligned}
\mathbb{E}e^{it\bar{X}_n} &= \mathbb{E}\exp\left\{ i\frac{t}{n}\sum_{j=1}^n X_j \right\} \\
&= \mathbb{E}\prod_{j=1}^n \exp\left\{ i\frac{t}{n} X_j \right\} \\
&= \prod_{j=1}^n \mathbb{E}\exp\left\{ i\frac{t}{n} X_j \right\} = [\phi(t/n)]^n
\end{aligned}$$

In view of (4.21), 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

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 μ and common variance $\sigma^2 \in (0, \infty)$, then

$$\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} N(0, \sigma^2) \quad \text{as} \quad n \to \infty \tag{4.22}$$

Here $\xrightarrow{d} N(0, \sigma^2)$ indicates convergence in distribution to a centered (i.e, zero mean) normal with standard deviation σ

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 $\mathbb{P}\{X_i = 0\} = \mathbb{P}\{X_i = 1\} = 0.5$, 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 $n = 1, 2, 4, 8$

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)

```
"""
Filename: illustrates_clt.py
Authors: John Stachurski and Thomas J. Sargent

Visual illustration of the central limit theorem.  Histograms draws of

    Y_n := \sqrt{n} (\bar X_n - \mu)

for a given distribution of X_i, and a given choice of n.
"""
import numpy as np
from scipy.stats import expon, norm
import matplotlib.pyplot as plt
from matplotlib import rc

# == Specifying font, needs LaTeX integration == #
rc('font', **{'family': 'serif', 'serif': ['Palatino']})
rc('text', usetex=True)

# == Set parameters == #
n = 250      # Choice of n
k = 100000   # 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 == #
sample_means = data.mean(axis=1)
# == Generate observations of Y_n == #
Y = np.sqrt(n) * (sample_means - mu)

# == Plot == #
fig, ax = plt.subplots()
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=r'$N(0, \sigma^2)$
↪')
ax.legend()

plt.show()
```

The file is `illustrates_clt.py`, and can be downloaded here

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

Note: You might need to delete or modify the lines beginning with `rc` to get this code to run on your computer

Simulation 2

Our next simulation is somewhat like the first, except that we aim to track the distribution of $Y_n := \sqrt{n}(\bar{X}_n - \mu)$ as n increases

In the simulation we'll be working with random variables having $\mu = 0$

Thus, when $n = 1$, we have $Y_1 = X_1$, so the first distribution is just the distribution of the underlying random variable

For $n = 2$, the distribution of Y_2 is that of $(X_1 + X_2)/\sqrt{2}$, and so on

What we expect is that, regardless of the distribution of the underlying random variable, the distribution of Y_n will smooth out into a bell shaped curve

The next figure shows this process for $X_i \sim f$, where f was specified as the convex combination of three different beta densities

(Taking a convex combination is an easy way to produce an irregular shape for f)

In the figure, the closest density is that of Y_1, while the furthest is that of Y_5

As expected, the distribution smooths out into a bell curve as n increases

The figure is generated by file `clt3d.py`, and can be downloaded here

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

$$\mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1, \ldots, \mathbf{X}_n \leq \mathbf{x}_n\} = \mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1\} \times \cdots \times \mathbb{P}\{\mathbf{X}_n \leq \mathbf{x}_n\}$$

(The vector inequality $\mathbf{X} \leq \mathbf{x}$ means that $X_j \leq x_j$ for $j = 1, \ldots, k$)

Let $\mu_j := \mathbb{E}[X_j]$ for all $j = 1, \ldots, k$

The expectation $\mathbb{E}[\mathbf{X}]$ of $\mathbf{X}$ is defined to be the vector of expectations:

$$\mathbb{E}[\mathbf{X}] := \begin{pmatrix} \mathbb{E}[X_1] \\ \mathbb{E}[X_2] \\ \vdots \\ \mathbb{E}[X_k] \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_k \end{pmatrix} =: \boldsymbol{\mu}$$

The *variance-covariance matrix* of random vector $\mathbf{X}$ is defined as

$$\mathrm{Var}[\mathbf{X}] := \mathbb{E}[(\mathbf{X} - \boldsymbol{\mu})(\mathbf{X} - \boldsymbol{\mu})']$$

Expanding this out, we get

$$\mathrm{Var}[\mathbf{X}] = \begin{pmatrix} \mathbb{E}[(X_1 - \mu_1)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_1 - \mu_1)(X_k - \mu_k)] \\ \mathbb{E}[(X_2 - \mu_2)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_2 - \mu_2)(X_k - \mu_k)] \\ \vdots & \vdots & \vdots \\ \mathbb{E}[(X_k - \mu_k)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_k - \mu_k)(X_k - \mu_k)] \end{pmatrix}$$

The j, k-th term is the scalar covariance between X_j and X_k

With this notation we can proceed to the multivariate LLN and CLT

Let $\mathbf{X}_1, \ldots, \mathbf{X}_n$ be a sequence of independent and identically distributed random vectors, each one taking values in $\mathbb{R}^k$

Let $\boldsymbol{\mu}$ be the vector $\mathbb{E}[\mathbf{X}_i]$, and let Σ be the variance-covariance matrix of $\mathbf{X}_i$

Interpreting vector addition and scalar multiplication in the usual way (i.e., pointwise), let

$$\bar{\mathbf{X}}_n := \frac{1}{n} \sum_{i=1}^{n} \mathbf{X}_i$$

In this setting, the LLN tells us that

$$\mathbb{P}\left\{\bar{\mathbf{X}}_n \to \boldsymbol{\mu} \text{ as } n \to \infty\right\} = 1 \tag{4.23}$$

Here $\bar{\mathbf{X}}_n \to \boldsymbol{\mu}$ means that $\|\bar{\mathbf{X}}_n \to \boldsymbol{\mu}\| \to 0$, where $\|\cdot\|$ is the standard Euclidean norm

The CLT tells us that, provided Σ is finite,

$$\sqrt{n}(\bar{\mathbf{X}}_n - \boldsymbol{\mu}) \xrightarrow{d} N(\mathbf{0}, \Sigma) \quad \text{as} \quad n \to \infty \tag{4.24}$$

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 $g\colon \mathbb{R} \to \mathbb{R}$ is differentiable at μ and $g'(\mu) \neq 0$, then

$$\sqrt{n}\{g(\bar{X}_n) - g(\mu)\} \xrightarrow{d} N(0, g'(\mu)^2\sigma^2) \quad \text{as} \quad n \to \infty \tag{4.25}$$

This theorem is used frequently in statistics to obtain the asymptotic distribution of estimators — many of which can be expressed as functions of sample means

(These kinds of results are often said to use the "delta method")

The proof is based on a Taylor expansion of g around the point μ

Taking the result as given, let the distribution F of each X_i be uniform on $[0, \pi/2]$ and let $g(x) = \sin(x)$

Derive the asymptotic distribution of $\sqrt{n}\{g(\bar{X}_n) - g(\mu)\}$ and illustrate convergence in the same spirit as the program `illustrate_clt.py` discussed above

What happens when you replace $[0, \pi/2]$ with $[0, \pi]$?

What is the source of the problem?

Exercise 2

Here's a result that's often used in developing statistical tests, and is connected to the multivariate central limit theorem

If you study econometric theory, you will see this result used again and again

Assume the setting of the multivariate CLT *discussed above*, so that

1. $\mathbf{X}_1, \ldots, \mathbf{X}_n$ is a sequence of IID random vectors, each taking values in $\mathbb{R}^k$
2. $\boldsymbol{\mu} := \mathbb{E}[\mathbf{X}_i]$, and Σ is the variance-covariance matrix of $\mathbf{X}_i$
3. The convergence

$$\sqrt{n}(\bar{\mathbf{X}}_n - \boldsymbol{\mu}) \xrightarrow{d} N(\mathbf{0}, \Sigma) \tag{4.26}$$

is valid

In a statistical setting, one often wants the right hand side to be **standard** normal, so that confidence intervals are easily computed

This normalization can be achieved on the basis of three observations

First, if $\mathbf{X}$ is a random vector in $\mathbb{R}^k$ and $\mathbf{A}$ is constant and $k \times k$, then

$$\operatorname{Var}[\mathbf{A}\mathbf{X}] = \mathbf{A} \operatorname{Var}[\mathbf{X}]\mathbf{A}'$$

Second, by the continuous mapping theorem, if $\mathbf{Z}_n \xrightarrow{d} \mathbf{Z}$ in $\mathbb{R}^k$ and $\mathbf{A}$ is constant and $k \times k$, then

$$\mathbf{A}\mathbf{Z}_n \xrightarrow{d} \mathbf{A}\mathbf{Z}$$

Third, if $\mathbf{S}$ is a $k \times k$ symmetric positive definite matrix, then there exists a symmetric positive definite matrix $\mathbf{Q}$, called the inverse square root of $\mathbf{S}$, such that

$$\mathbf{Q}\mathbf{S}\mathbf{Q}' = \mathbf{I}$$

Here $\mathbf{I}$ is the $k \times k$ identity matrix

Putting these things together, your first exercise is to show that if $\mathbf{Q}$ is the inverse square root of $\boldsymbol{\Sigma}$, then

$$\mathbf{Z}_n := \sqrt{n}\mathbf{Q}(\bar{\mathbf{X}}_n - \boldsymbol{\mu}) \xrightarrow{d} \mathbf{Z} \sim N(\mathbf{0}, \mathbf{I})$$

Applying the continuous mapping theorem one more time tells us that

$$\|\mathbf{Z}_n\|^2 \xrightarrow{d} \|\mathbf{Z}\|^2$$

Given the distribution of $\mathbf{Z}$, we conclude that

$$n\|\mathbf{Q}(\bar{\mathbf{X}}_n - \boldsymbol{\mu})\|^2 \xrightarrow{d} \chi^2(k) \tag{4.27}$$

where $\chi^2(k)$ is the chi-squared distribution with k degrees of freedom

(Recall that k is the dimension of $\mathbf{X}_i$, the underlying random vectors)

Your second exercise is to illustrate the convergence in (4.27) with a simulation

In doing so, let

$$\mathbf{X}_i := \begin{pmatrix} W_i \\ U_i + W_i \end{pmatrix}$$

where

- each W_i is an IID draw from the uniform distribution on $[-1, 1]$
- each U_i is an IID draw from the uniform distribution on $[-2, 2]$
- U_i and W_i are independent of each other

Hints:

1. `scipy.linalg.sqrtm(A)` computes the square root of `A`. You still need to invert it
2. You should be able to work out Σ from the proceding information

Solutions

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

Exercise 1

Here is one solution

You might have to modify or delete the lines starting with `rc`, depending on your configuration

```
"""
Illustrates the delta method, a consequence of the central limit theorem.
"""

from scipy.stats import uniform, norm
from matplotlib import rc

# == Specifying font, needs LaTeX integration == #
rc('font',**{'family':'serif','serif':['Palatino']})
rc('text', usetex=True)

# == Set parameters == #
n = 250
replications = 100000
distribution = uniform(loc=0, scale=(np.pi / 2))
mu, s = distribution.mean(), distribution.std()

g = np.sin
g_prime = np.cos

# == Generate obs of sqrt{n} (g(\bar X_n) - g(\mu)) == #
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(mu))

# == Plot == #
asymptotic_sd = g_prime(mu) * s
fig, ax = plt.subplots(figsize=(10, 6))
xmin = -3 * g_prime(mu) * 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 = r"$N(0, g'(\mu)^2  \sigma^2)$"
ax.plot(xgrid, norm.pdf(xgrid, scale=asymptotic_sd), 'k-', lw=2, label=lb)
ax.legend()
plt.show()
```

What happens when you replace $[0, \pi/2]$ with $[0, \pi]$?

In this case, the mean μ of this distribution is $\pi/2$, and since $g' = \cos$, we have $g'(\mu) = 0$

Hence the conditions of the delta theorem are not satisfied

Exercise 2

First we want to verify the claim that

$$\sqrt{n}\mathbf{Q}(\bar{\mathbf{X}}_n - \boldsymbol{\mu}) \xrightarrow{d} N(\mathbf{0}, \mathbf{I})$$

This is straightforward given the facts presented in the exercise

Let

$$\mathbf{Y}_n := \sqrt{n}(\bar{\mathbf{X}}_n - \boldsymbol{\mu}) \quad \text{and} \quad \mathbf{Y} \sim N(\mathbf{0}, \Sigma)$$

By the multivariate CLT and the continuous mapping theorem, we have

$$\mathbf{Q}\mathbf{Y}_n \xrightarrow{d} \mathbf{Q}\mathbf{Y}$$

Since linear combinations of normal random variables are normal, the vector $\mathbf{Q}\mathbf{Y}$ is also normal

Its mean is clearly $\mathbf{0}$, and its variance covariance matrix is

$$\mathrm{Var}[\mathbf{Q}\mathbf{Y}] = \mathbf{Q}\mathrm{Var}[\mathbf{Y}]\mathbf{Q}' = \mathbf{Q}\Sigma\mathbf{Q}' = \mathbf{I}$$

In conclusion, $\mathbf{QY}_n \xrightarrow{d} \mathbf{QY} \sim N(\mathbf{0}, \mathbf{I})$, which is what we aimed to show

Now we turn to the simulation exercise

Our solution is as follows

```
from scipy.stats import uniform, 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
Sigma = ((vw, vw), (vw, vw + vu))
Sigma = np.array(Sigma)

# == Compute Sigma^{-1/2} == #
Q = inv(sqrtm(Sigma))

# == 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 = np.dot(Q, error_obs)
chisq_obs = np.sum(temp**2, axis=0)

# == Plot == #
fig, ax = plt.subplots(figsize=(10, 6))
xmax = 8
ax.set_xlim(0, xmax)
xgrid = np.linspace(0, xmax, 200)
lb = "Chi-squared with 2 degrees of freedom"
ax.plot(xgrid, chi2.pdf(xgrid, 2), 'k-', lw=2, label=lb)
ax.legend()
ax.hist(chisq_obs, bins=50, normed=True)
plt.show()
```

Linear State Space Models

Contents

- *Linear State Space Models*
  - *Overview*
  - *The Linear State Space Model*
  - *Distributions and Moments*
  - *Stationarity and Ergodicity*
  - *Noisy Observations*
  - *Prediction*
  - *Code*
  - *Exercises*
  - *Solutions*

"We may regard the present state of the universe as the effect of its past and the cause of its future" – Marquis de Laplace

Overview

This lecture introduces the linear state space dynamic system

Easy to use and carries a powerful theory of prediction

A workhorse with many applications

- 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., etc., . . .
- key ingredient of useful models
  - Friedman's permanent income model of consumption smoothing
  - Barro's model of smoothing total tax collections
  - Rational expectations version of Cagan's model of hyperinflation
  - Sargent and Wallace's "unpleasant monetarist arithmetic"
  - etc., etc., . . .

The Linear State Space Model

Objects in play

- An $n \times 1$ vector x_t denoting the **state** at time $t = 0, 1, 2, \ldots$
- An iid sequence of $m \times 1$ random vectors $w_t \sim N(0, I)$
- A $k \times 1$ vector y_t of **observations** at time $t = 0, 1, 2, \ldots$
- An $n \times n$ matrix A called the **transition matrix**
- An $n \times m$ matrix C called the **volatility matrix**
- A $k \times n$ matrix G sometimes called the **output matrix**

Here is the linear state-space system

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t \\ x_0 &\sim N(\mu_0, \Sigma_0) \end{aligned}$$

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 (4.28) pins down the values of the sequences $\{x_t\}$ and $\{y_t\}$

Even without these draws, the primitives 1–3 pin down the *probability distributions* of $\{x_t\}$ and $\{y_t\}$

Later we'll see how to compute these distributions and their moments

Martingale difference shocks

We've made the common assumption that the shocks are independent standardized normal vectors

But some of what we say will be valid under the assumption that $\{w_{t+1}\}$ is a **martingale difference sequence**

A martingale difference sequence is a sequence that is zero mean when conditioned on past information

In the present case, since $\{x_t\}$ is our state sequence, this means that it satisfies

$$\mathbb{E}[w_{t+1}|x_t, x_{t-1}, \ldots] = 0$$

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 satifies

$$y_{t+1} = \phi_0 + \phi_1 y_t + \phi_2 y_{t-1} \quad \text{s.t.} \quad y_0, y_{-1} \text{ given} \tag{4.28}$$

To map (4.28) into our state space system (4.28), we set

$$x_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix} \qquad A = \begin{bmatrix} 1 & 0 & 0 \\ \phi_0 & \phi_1 & \phi_2 \\ 0 & 1 & 0 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

You can confirm that under these definitions, (4.28) and (4.28) 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 (4.28) to represent the model

$$y_{t+1} = \phi_1 y_t + \phi_2 y_{t-1} + \phi_3 y_{t-2} + \phi_4 y_{t-3} + \sigma w_{t+1} \tag{4.29}$$

where $\{w_t\}$ is iid and standard normal

To put this in the linear state space format we take $x_t = \begin{bmatrix} y_t & y_{t-1} & y_{t-2} & y_{t-3} \end{bmatrix}'$ and

$$A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

The matrix A has the form of the *companion matrix* to the vector $\begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \end{bmatrix}$.

The next figure shows dynamics of this process when

$$\phi_1 = 0.5, \phi_2 = -0.2, \phi_3 = 0, \phi_4 = 0.5, \sigma = 0.2, y_0 = y_{-1} = y_{-2} = y_{-3} = 1$$

Vector Autoregressions

Now suppose that

- y_t is a $k \times 1$ vector
- ϕ_j is a $k \times k$ matrix and
- w_t is $k \times 1$

Then (4.29) is termed a *vector autoregression*

To map this into (4.28), we set

$$x_t = \begin{bmatrix} y_t \\ y_{t-1} \\ y_{t-2} \\ y_{t-3} \end{bmatrix} \quad A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \end{bmatrix} \quad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad G = \begin{bmatrix} I & 0 & 0 & 0 \end{bmatrix}$$

where I is the $k \times k$ identity matrix and σ is a $k \times k$ matrix

Seasonals

We can use (4.28) 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 (4.29)

With the deterministic seasonal, the transition matrix becomes

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

It is easy to check that $A^4 = I$, which implies that x_t is strictly periodic with period 4:[1]

$$x_{t+4} = x_t$$

Such an x_t process can be used to model deterministic seasonals in quarterly time series.

The *indeterministic* seasonal produces recurrent, but aperiodic, seasonal fluctuations.

Time Trends

The model $y_t = at + b$ is known as a *linear time trend*

We can represent this model in the linear state space form by taking

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} a & b \end{bmatrix} \tag{4.30}$$

and starting at initial condition $x_0 = \begin{bmatrix} 0 & 1 \end{bmatrix}'$

In fact it's possible to use the state-space system to represent polynomial trends of any order

For instance, let

$$x_0 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \qquad A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

It follows that

$$A^t = \begin{bmatrix} 1 & t & t(t-1)/2 \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix}$$

Then $x_t' = \begin{bmatrix} t(t-1)/2 & t & 1 \end{bmatrix}$, 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 (4.28) repeatedly to obtain

$$\begin{aligned} x_t &= Ax_{t-1} + Cw_t \\ &= A^2 x_{t-2} + ACw_{t-1} + Cw_t \\ &\vdots \\ &= \sum_{j=0}^{t-1} A^j C w_{t-j} + A^t x_0 \end{aligned}$$

Representation (4.31) is a *moving average* representation

It expresses $\{x_t\}$ as a linear function of

1. current and past values of the process $\{w_t\}$ and
2. the initial condition x_0

As an example of a moving average representation, let the model be

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

You will be able to show that $A^t = \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix}$ and $A^j C = \begin{bmatrix} 1 & 0 \end{bmatrix}'$

Substituting into the moving average representation (4.31), 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*

Distributions and Moments

Unconditional Moments

Using (4.28), it's easy to obtain expressions for the (unconditional) means of x_t and y_t

We'll explain what *unconditional* and *conditional* mean soon

Letting $\mu_t := \mathbb{E}[x_t]$ and using linearity of expectations, we find that

$$\mu_{t+1} = A\mu_t \quad \text{with} \quad \mu_0 \text{ given} \tag{4.31}$$

Here μ_0 is a primitive given in (4.28)

The variance-covariance matrix of x_t is $\Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)']$

Using $x_{t+1} - \mu_{t+1} = A(x_t - \mu_t) + Cw_{t+1}$, we can determine this matrix recursively via

$$\Sigma_{t+1} = A\Sigma_t A' + CC' \quad \text{with} \quad \Sigma_0 \text{ given} \tag{4.32}$$

As with μ_0, the matrix Σ_0 is a primitive given in (4.28)

As a matter of terminology, we will sometimes call

- μ_t the *unconditional mean* of x_t
- Σ_t the *unconditional variance-convariance matrix* of x_t

This is to distinguish μ_t and Σ_t from related objects that use conditioning information, to be defined below

However, you should be aware that these "unconditional" moments do depend on the initial distribution $N(\mu_0, \Sigma_0)$

Moments of the Observations

Using linearity of expectations again we have

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t] = G\mu_t \tag{4.33}$$

The variance-covariance matrix of y_t is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t] = G\Sigma_t G' \tag{4.34}$$

Distributions

In general, knowing the mean and variance-covariance matrix of a random vector is not quite as good as knowing the full distribution

However, there are some situations where these moments alone tell us all we need to know

These are situations in which the mean vector and covariance matrix are **sufficient statistics** for the population distribution

(Sufficient statistics form a list of objects that characterize a population distribution)

One such situation is when the vector in question is Gaussian (i.e., normally distributed)

This is the case here, given

1. our Gaussian assumptions on the primitives
2. the fact that normality is preserved under linear operations

In fact, it's well-known that

$$u \sim N(\bar{u}, S) \quad \text{and} \quad v = a + Bu \implies v \sim N(a + B\bar{u}, BSB') \tag{4.35}$$

In particular, given our Gaussian assumptions on the primitives and the linearity of (4.28) we can see immediately that both x_t and y_t are Gaussian for all $t \geq 0$[2]

Since x_t is Gaussian, to find the distribution, all we need to do is find its mean and variance-covariance matrix

But in fact we've already done this, in (4.31) and (4.32)

Letting μ_t and Σ_t be as defined by these equations, we have

$$x_t \sim N(\mu_t, \Sigma_t) \tag{4.36}$$

By similar reasoning combined with (4.33) and (4.34),

$$y_t \sim N(G\mu_t, G\Sigma_t G') \tag{4.37}$$

Ensemble Interpretations

How should we interpret the distributions defined by (4.36)–(4.37)?

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 (4.29)

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

[2] The correct way to argue this is by induction. Suppose that x_t is Gaussian. Then (4.28) and (4.35) 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.

(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 (4.37)

The histogram and population distribution are close, as expected

By looking at the figures and experimenting with parameters, you will gain a feel for how the population

distribution depends on the model primitives *listed above*, as intermediated by the distribution's sufficient statistics

Ensemble means

In the preceding figure we approximated the population distribution of y_T by

1. generating I sample paths (i.e., time series) where I is a large number
2. recording each observation y_T^i
3. histogramming this sample

Just as the histogram approximates the population distribution, the *ensemble* or *cross-sectional average*

$$\bar{y}_T := \frac{1}{I} \sum_{i=1}^{I} y_T^i$$

approximates the expectation $\mathbb{E}[y_T] = G\mu_T$ (as implied by the law of large numbers)

Here's a simulation comparing the ensemble averages and population means at time points $t = 0, \ldots, 50$

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

$$\bar{x}_T := \frac{1}{I} \sum_{i=1}^{I} x_T^i \to \mu_T \qquad (I \to \infty)$$

The limit μ_T is a "long-run average"

(By *long-run average* we mean the average for an infinite ($I = \infty$) number of sample x_T's)

Another application of the law of large numbers assures us that

$$\frac{1}{I}\sum_{i=1}^{I}(x_T^i - \bar{x}_T)(x_T^i - \bar{x}_T)' \to \Sigma_T \qquad (I \to \infty)$$

Joint Distributions

In the preceding discussion we looked at the distributions of x_t and y_t in isolation

This gives us useful information, but doesn't allow us to answer questions like

- what's the probability that $x_t \geq 0$ for all t?
- what's the probability that the process $\{y_t\}$ exceeds some value a before falling below b?
- etc., etc.

Such questions concern the *joint distributions* of these sequences

To compute the joint distribution of $x_0, x_1, \ldots, x_T$, recall that joint and conditional densities are linked by the rule

$$p(x, y) = p(y \,|\, x)p(x) \qquad \text{(joint } = \text{ conditional } \times \text{ marginal)}$$

From this rule we get $p(x_0, x_1) = p(x_1 \,|\, x_0)p(x_0)$

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 (4.28), 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*

$$\Sigma_{t+j,t} := \mathbb{E}[(x_{t+j} - \mu_{t+j})(x_t - \mu_t)'] \tag{4.38}$$

Elementary calculations show that

$$\Sigma_{t+j,t} = A^j \Sigma_t \tag{4.39}$$

Notice that $\Sigma_{t+j,t}$ in general depends on both j, the gap between the two dates, and t, the earlier date

Stationarity and Ergodicity

Stationarity and ergodicity are two properties that, when they hold, greatly aid analysis of linear state space models

Let's start with the intuition

Visualizing Stability

Let's look at some more time series from the same model that we analyzed above

This picture shows cross-sectional distributions for y at times T, T', T''

Note how the time series "settle down" in the sense that the distributions at T' and T'' are relatively similar to each other — but unlike the distribution at T

Apparently, the distributions of y_t converge to a fixed long-run distribution as $t \to \infty$

When such a distribution exists it is called a *stationary distribution*

Stationary Distributions

In our setting, a distribution ψ_∞ is said to be *stationary* for x_t if

$$x_t \sim \psi_\infty \quad \text{and} \quad x_{t+1} = Ax_t + Cw_{t+1} \quad \Longrightarrow \quad x_{t+1} \sim \psi_\infty$$

Since

1. in the present case all distributions are Gaussian
2. a Gaussian distribution is pinned down by its mean and variance-covariance matrix

we can restate the definition as follows: ψ_∞ is stationary for x_t if

$$\psi_\infty = N(\mu_\infty, \Sigma_\infty)$$

where μ_∞ and Σ_∞ are fixed points of (4.31) and (4.32) 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 μ_∞ and Σ_∞ as fixed points of (4.31) and (4.32) 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 (4.39), the autocovariance function takes the form $\Sigma_{t+j,t} = A^j \Sigma_\infty$, which depends on j but not on t

This motivates the following definition

A process $\{x_t\}$ is said to be *covariance stationary* if

- both μ_t and Σ_t are constant in t
- $\Sigma_{t+j,t}$ depends on the time gap j but not on time t

In our setting, $\{x_t\}$ will be covariance stationary if μ_0, Σ_0, A, C assume values that imply that none of $\mu_t, \Sigma_t, \Sigma_{t+j,t}$ depends on t

Conditions for Stationarity

The globally stable case

The difference equation $\mu_{t+1} = A\mu_t$ is known to have *unique* fixed point $\mu_\infty = 0$ if all eigenvalues of A have moduli strictly less than unity

That is, if `(np.absolute(np.linalg.eigvals(A)) < 1).all() == True`

The difference equation (4.32) also has a unique fixed point in this case, and, moreover

$$\mu_t \to \mu_\infty = 0 \quad \text{and} \quad \Sigma_t \to \Sigma_\infty \quad \text{as} \quad t \to \infty$$

regardless of the initial conditions μ_0 and Σ_0

This is the *globally stable case* — see these notes for more a theoretical treatment

However, global stability is more than we need for stationary solutions, and often more than we want

To illustrate, consider *our second order difference equation example*

Here the state is $x_t = \begin{bmatrix} 1 & y_t & y_{t-1} \end{bmatrix}'$

Because of the constant first component in the state vector, we will never have $\mu_t \to 0$

How can we find stationary solutions that respect a constant state component?

Processes with a constant state component

To investigate such a process, suppose that A and C take the form

$$A = \begin{bmatrix} A_1 & a \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} C_1 \\ 0 \end{bmatrix}$$

where

- A_1 is an $(n-1) \times (n-1)$ matrix
- a is an $(n-1) \times 1$ column vector

Let $x_t = \begin{bmatrix} x'_{1t} & 1 \end{bmatrix}'$ where x_{1t} is $(n-1) \times 1$

It follows that

$$x_{1,t+1} = A_1 x_{1t} + a + C_1 w_{t+1}$$

Let $\mu_{1t} = \mathbb{E}[x_{1t}]$ and take expectations on both sides of this expression to get

$$\mu_{1,t+1} = A_1 \mu_{1,t} + a \tag{4.40}$$

Assume now that the moduli of the eigenvalues of A_1 are all strictly less than one

Then (4.40) has a unique stationary solution, namely,

$$\mu_{1\infty} = (I - A_1)^{-1} a$$

The stationary value of μ_t itself is then $\mu_\infty := \begin{bmatrix} \mu'_{1\infty} & 1 \end{bmatrix}'$

The stationary values of Σ_t and $\Sigma_{t+j,t}$ satisfy

$$\Sigma_\infty = A \Sigma_\infty A' + CC'$$
$$\Sigma_{t+j,t} = A^j \Sigma_\infty$$

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 (4.32) converge to the fixed point of the *discrete Lyapunov equation* in the first line of (4.41)

Ergodicity

Let's suppose that we're working with a covariance stationary process

In this case we know that the ensemble mean will converge to μ_∞ as the sample size I approaches infinity

Averages over time

Ensemble averages across simulations are interesting theoretically, but in real life we usually observe only a *single* realization $\{x_t, y_t\}_{t=0}^T$

So now let's take a single realization and form the time series averages

$$\bar{x} := \frac{1}{T} \sum_{t=1}^T x_t \quad \text{and} \quad \bar{y} := \frac{1}{T} \sum_{t=1}^T y_t$$

Do these time series averages converge to something interpretable in terms of our basic state-space representation?

The answer depends on something called *ergodicity*

Ergodicity is the property that time series and ensemble averages coincide

More formally, ergodicity implies that time series sample averages converge to their expectation under the stationary distribution

In particular,

- $\frac{1}{T}\sum_{t=1}^{T} x_t \to \mu_\infty$
- $\frac{1}{T}\sum_{t=1}^{T}(x_t - \bar{x}_T)(x_t - \bar{x}_T)' \to \Sigma_\infty$
- $\frac{1}{T}\sum_{t=1}^{T}(x_{t+j} - \bar{x}_T)(x_t - \bar{x}_T)' \to A^j\Sigma_\infty$

In our linear Gaussian setting, any covariance stationary process is also ergodic

Noisy Observations

In some settings the observation equation $y_t = Gx_t$ is modified to include an error term

Often this error term represents the idea that the true state can only be observed imperfectly

To include an error term in the observation we introduce

- An iid sequence of $\ell \times 1$ random vectors $v_t \sim N(0, I)$
- A $k \times \ell$ matrix H

and extend the linear state-space system to

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + Hv_t \\ x_0 &\sim N(\mu_0, \Sigma_0) \end{aligned}$$

The sequence $\{v_t\}$ is assumed to be independent of $\{w_t\}$

The process $\{x_t\}$ is not modified by noise in the observation equation and its moments, distributions and stability properties remain the same

The unconditional moments of y_t from (4.33) and (4.34) now become

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t + Hv_t] = G\mu_t \tag{4.41}$$

The variance-covariance matrix of y_t is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t + Hv_t] = G\Sigma_t G' + HH' \tag{4.42}$$

The distribution of y_t is therefore

$$y_t \sim N(G\mu_t, G\Sigma_t G' + HH')$$

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

$$\mathbb{E}_t[x_{t+1}] := \mathbb{E}[x_{t+1} \mid x_t, x_{t-1}, \ldots, x_0] = Ax_t$$

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 $\mathbb{E}_t[x_{t+1}] = \mathbb{E}[x_{t+1} \mid x_t]$ is an implication of $\{x_t\}$ having the *Markov property*

The one-step-ahead forecast error is

$$x_{t+1} - \mathbb{E}_t[x_{t+1}] = Cw_{t+1}$$

The covariance matrix of the forecast error is

$$\mathbb{E}[(x_{t+1} - \mathbb{E}_t[x_{t+1}])(x_{t+1} - \mathbb{E}_t[x_{t+1}])'] = CC'$$

More generally, we'd like to compute the j-step ahead forecasts $\mathbb{E}_t[x_{t+j}]$ and $\mathbb{E}_t[y_{t+j}]$

With a bit of algebra we obtain

$$x_{t+j} = A^j x_t + A^{j-1}Cw_{t+1} + A^{j-2}Cw_{t+2} + \cdots + A^0 Cw_{t+j}$$

In view of the iid property, current and past state values provide no information about future values of the shock

Hence $\mathbb{E}_t[w_{t+k}] = \mathbb{E}[w_{t+k}] = 0$

It now follows from linearity of expectations that the j-step ahead forecast of x is

$$\mathbb{E}_t[x_{t+j}] = A^j x_t$$

The j-step ahead forecast of y is therefore

$$\mathbb{E}_t[y_{t+j}] = \mathbb{E}_t[Gx_{t+j} + Hv_{t+j}] = GA^j x_t$$

Covariance of Prediction Errors

It is useful to obtain the covariance matrix of the vector of j-step-ahead prediction errors

$$x_{t+j} - \mathbb{E}_t[x_{t+j}] = \sum_{s=0}^{j-1} A^s Cw_{t-s+j} \tag{4.43}$$

Evidently,

$$V_j := \mathbb{E}_t[(x_{t+j} - \mathbb{E}_t[x_{t+j}])(x_{t+j} - \mathbb{E}_t[x_{t+j}])'] = \sum_{k=0}^{j-1} A^k C C' A^{k'} \tag{4.44}$$

V_j defined in (4.44) can be calculated recursively via $V_1 = CC'$ and

$$V_j = CC' + AV_{j-1}A', \quad j \geq 2 \tag{4.45}$$

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

$$V_\infty = CC' + AV_\infty A' \tag{4.46}$$

Equation (4.46) is an example of a *discrete Lyapunov* equation in the covariance matrix V_∞

A sufficient condition for V_j to converge is that the eigenvalues of A be strictly less than one in modulus.

Weaker sufficient conditions for convergence associate eigenvalues equaling or exceeding one in modulus with elements of C that equal 0

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 (4.28)

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

$$\mathbb{E}_t \left[\sum_{j=0}^{\infty} \beta^j y_{t+j} \right] = G[I + \beta A + \beta^2 A^2 + \cdots] x_t = G[I - \beta A]^{-1} x_t$$

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

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

Solutions

```
import numpy as np
import matplotlib.pyplot as plt
from quantecon import LinearStateSpace
```

Exercise 1

```
phi_0, phi_1, phi_2 = 1.1, 0.8, -0.8

A = [[1,     0,     0],
     [phi_0, phi_1, phi_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=(8, 4.6))
y = y.flatten()
ax.plot(y, 'b-', lw=2, alpha=0.7)
ax.grid()
ax.set_xlabel('time')
ax.set_ylabel(r'$y_t$', fontsize=16)
plt.show()
```

Exercise 2

```
phi_1, phi_2, phi_3, phi_4 = 0.5, -0.2, 0, 0.5
sigma = 0.2

A = [[phi_1, phi_2, phi_3, phi_4],
     [1,     0,     0,     0],
     [0,     1,     0,     0],
     [0,     0,     1,     0]]
C = [[sigma],
     [0],
     [0],
     [0]]
G = [1, 0, 0, 0]

ar = LinearStateSpace(A, C, G, mu_0=np.ones(4))
x, y = ar.simulate(ts_length=200)

fig, ax = plt.subplots(figsize=(8, 4.6))
y = y.flatten()
ax.plot(y, 'b-', lw=2, alpha=0.7)
ax.grid()
ax.set_xlabel('time')
ax.set_ylabel(r'$y_t$', fontsize=16)
plt.show()
```

Exercise 3

```
from scipy.stats import norm
import random

phi_1, phi_2, phi_3, phi_4 = 0.5, -0.2, 0, 0.5
sigma = 0.1

A = [[phi_1, phi_2, phi_3, phi_4],
     [1,     0,     0,     0],
     [0,     1,     0,     0],
     [0,     0,     1,     0]]
C = [[sigma],
     [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(r'time', fontsize=16)
ax.set_ylabel(r'$y_t$', 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=r'$\bar y_t$')

m = ar.moment_sequence()
population_means = []
for t in range(T):
    mu_x, mu_y, Sigma_x, Sigma_y = next(m)
    population_means.append(float(mu_y))
ax.plot(population_means, color='g', lw=2, alpha=0.8, label=r'$G\mu_t$')
ax.legend(ncol=2)
plt.show()
```

Exercise 4

```
phi_1, phi_2, phi_3, phi_4 = 0.5, -0.2, 0, 0.5
sigma = 0.1
```

```
A = [[phi_1, phi_2, phi_3, phi_4],
     [1,     0,     0,     0],
     [0,     1,     0,     0],
     [0,     0,     1,     0]]
C = [[sigma],
     [0],
     [0],
     [0]]
G = [1, 0, 0, 0]

T0 = 10
T1 = 50
T2 = 75
T4 = 100

ar = LinearStateSpace(A, C, G, mu_0=np.ones(4))
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(r'$y_t$', fontsize=16)
ax.vlines((T0, T1, T2), -1.5, 1.5)

ax.set_xticks((T0, T1, T2))
ax.set_xticklabels((r"$T$", r"$T'$", r"$T''$"), 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()
```

0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
y_t
T
T'
T''

0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
y_t
T
T'
T''

Continuous State Markov Chains

Contents

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

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 we'll see *later on*

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

In particular,

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

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

Note: For some people, the term "Markov chain" always refers to a process with a finite or discrete state space. We follow the mainstream mathematical literature (e.g., *[MT09]*) in using the term to refer to any discrete **time** Markov process

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 — a nonnegative square matrix $P = P[i, j]$ such that each row $P[i, \cdot]$ sums to one

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

In symbols,

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

Equivalently,

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

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

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

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

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

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

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

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

(Integrals are over the whole space unless otherwise specified)

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

$$p_w(x, y) := \frac{1}{\sqrt{2\pi}} \exp\left\{ -\frac{(y - x)^2}{2} \right\} \tag{4.47}$$

What kind of model does p_w represent?

The answer is, the (normally distributed) random walk

$$X_{t+1} = X_t + \xi_{t+1} \quad \text{where} \quad \{\xi_t\} \overset{\text{IID}}{\sim} N(0, 1) \tag{4.48}$$

To see this, let's find the stochastic kernel p corresponding to (4.48)

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

Letting $X_t = x$ in (4.48) 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 (4.47)

Connection to Stochastic Difference Equations

In the previous section, we made the connection between stochastic difference equation (4.48) and stochastic kernel (4.47)

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

$$X_{t+1} = \mu(X_t) + \sigma(X_t)\, \xi_{t+1} \tag{4.49}$$

Here we assume that

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

Example 1: The random walk (4.48) is a special case of (4.49), with $\mu(x) = x$ and $\sigma(x) = 1$

Example 2: Consider the ARCH model

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

Alternatively, we can write the model as

$$X_{t+1} = \alpha X_t + (\beta + \gamma X_t^2)^{1/2} \xi_{t+1} \tag{4.50}$$

This is a special case of (4.49) 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

$$k_{t+1} = s A_{t+1} f(k_t) + (1 - \delta) k_t \tag{4.51}$$

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
- δ is a depreciation rate
- $f \colon \mathbb{R}_+ \to \mathbb{R}_+$ is a production function satisfying $f(k) > 0$ whenever $k > 0$

(The fixed savings rate can be rationalized as the optimal policy for a particular set of technologies and preferences (see *[LS12]*, section 3.1.2), although we omit the details here)

Equation (4.51) is a special case of (4.49) with $\mu(x) = (1 - \delta)x$ and $\sigma(x) = s f(x)$

Now let's obtain the stochastic kernel corresponding to the generic model (4.49)

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

$$f_V(v) = \frac{1}{b} f_U \left(\frac{v - a}{b} \right) \tag{4.52}$$

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

Taking (4.52) as given for the moment, we can obtain the stochastic kernel p for (4.49) 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 (4.52),

$$p(x, y) = \frac{1}{\sigma(x)} \phi \left(\frac{y - \mu(x)}{\sigma(x)} \right) \tag{4.53}$$

For example, the growth model in (4.51) has stochastic kernel

$$p(x, y) = \frac{1}{s f(x)} \phi \left(\frac{y - (1 - \delta)x}{s f(x)} \right) \tag{4.54}$$

where ϕ is the density of A_{t+1}

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

Distribution Dynamics

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

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

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

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

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

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

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

$$\psi_{t+1}(y) = \int p(x, y) \psi_t(x) \, dx, \qquad \forall y \in S \tag{4.55}$$

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

$$(\psi P)(y) = \int p(x, y) \psi(x) dx \tag{4.56}$$

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., ψP instead of $P\psi$). This is a common convention, with the intention being to maintain the parallel with the finite case — see *here*

With this notation, we can write (4.55) more succinctly as $\psi_{t+1}(y) = (\psi_t P)(y)$ for all y, or, dropping the y and letting "=" indicate equality of functions,

$$\psi_{t+1} = \psi_t P \tag{4.57}$$

Equation (4.57) tells us that if we specify a distribution for ψ_0, then the entire sequence of future distributions can be obtained by iterating with P

It's interesting to note that (4.57) is a deterministic difference equation

Thus, by converting a stochastic difference equation such as (4.49) 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's useful to compute and study the sequences of densities generated by the model

One way to do this is to try to implement the iteration described by (4.56) and (4.57) using numerical integration

However, to produce ψP from ψ via (4.56), 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:

$$k_{t+1} = sA_{t+1}f(k_t) + (1-\delta)k_t \tag{4.58}$$

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

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

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

1. draw k_0 from the specified initial condition ψ_0
2. draw the shocks $A_1, \ldots, A_t$ from their specified density ϕ
3. compute k_t iteratively via (4.58)

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

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

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

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

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

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

$$\psi_t^n(y) = \frac{1}{n} \sum_{i=1}^n p(k_{t-1}^i, y) \tag{4.59}$$

where p is the growth model stochastic kernel in (4.54)

What is the justification for this slightly surprising estimator?

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

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

with probability one as $n \to \infty$

Here the first equality is by the definition of ψ_{t-1}, and the second is by (4.55)

We have just shown that our estimator $\psi_t^n(y)$ in (4.59) 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 (4.59) 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

An example of usage for the stochastic growth model *described above* can be found in stochasticgrowth.py

When run, the code produces a figure like this

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*)

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

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

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

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

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

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

The distribution dynamics in (4.55) are then replaced by

$$F_{t+1}(y) = \int G(x, y) F_t(dx) \tag{4.60}$$

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

The intuition behind (4.60) is essentially the same as for (4.55)

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*

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 (4.56), a density ψ^* on S is called *stationary* for P if it is a fixed point of the operator P

In other words,

$$\psi^*(y) = \int p(x, y)\psi^*(x)\, dx, \qquad \forall y \in S \tag{4.61}$$

As with the finite case, if ψ^* is stationary for P, and the distribution of X_0 is ψ^*, then, in view of (4.57), X_t will have this same distribution for all t

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

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

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

Even existence can fail very easily

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

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

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

$$\forall\, \psi \in \mathscr{D}, \quad \psi P^t \to \psi^* \quad \text{as} \quad t \to \infty \tag{4.62}$$

This combination of existence, uniqueness and global convergence in the sense of (4.62) is often referred to as *global stability*

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

$$\frac{1}{n}\sum_{t=1}^{n} h(X_t) \to \int h(x)\psi^*(x)dx \quad \text{as } n \to \infty \tag{4.63}$$

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

Note that the convergence in (4.63) does not depend on the distribution (or value) of X_0

This is actually very important for simulation — it means we can learn about ψ^* (i.e., approximate the right hand side of (4.63) 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 (4.62) — 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 ψ^*

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

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

Let's say that we have a model of the form (4.49) that is stable and ergodic

Let p be the corresponding stochastic kernel, as given in (4.53)

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

$$\psi_n^*(y) = \frac{1}{n} \sum_{t=1}^{n} p(X_t, y) \tag{4.64}$$

This is essentially the same as the look ahead estimator (4.59), except that now the observations we generate are a single time series, rather than a cross section

The justification for (4.64) 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 (4.63) and the equality on the right is by (4.61)

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

Exercises

Exercise 1

Consider the simple threshold autoregressive model

$$X_{t+1} = \theta |X_t| + (1 - \theta^2)^{1/2} \xi_{t+1} \qquad \text{where} \quad \{\xi_t\} \overset{\text{IID}}{\sim} N(0, 1) \tag{4.65}$$

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 ψ^* given by

$$\psi^*(y) = 2 \, \phi(y) \, \Phi \left[\frac{\theta y}{(1 - \theta^2)^{1/2}} \right] \tag{4.66}$$

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

As an exercise, compute the look ahead estimate of ψ^*, as defined in (4.64), and compare it with ψ^* in (4.66) 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 = beta(5, 5, scale=0.5, loc=i*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

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

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((r'$X$', r'$Y$', r'$Z$'), fontsize=16)
plt.show()
```

The three data sets are

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

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 (4.65)

We know that the distribution of X_t will converge to (4.66) 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

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

For each X_0 in this set,

1. Generate k time series of length n, each starting at X_0 and obeying (4.65)
2. Create a boxplot representing n distributions, where the t-th distribution shows the k observations of X_t

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

Solutions

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

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.

```
from scipy.stats import norm, gaussian_kde
from quantecon import LAE

phi = norm()
n = 500
theta = 0.8
# == Frequently used constants == #
d = np.sqrt(1 - theta**2)
delta = theta / d

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

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

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

fig, ax = plt.subplots(figsize=(10,7))
ys = np.linspace(-3, 3, 200)
ax.plot(ys, psi_star(ys), 'b-', lw=2, alpha=0.6, label='true')
ax.plot(ys, psi_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

```
from scipy.stats import lognorm, beta

# == Define parameters == #
s = 0.2
delta = 0.1
a_sigma = 0.4        # A = exp(B) where B ~ N(0, a_sigma)
alpha = 0.4          # f(k) = k**alpha

phi = lognorm(a_sigma)

def p(x, y):
    "Stochastic kernel, vectorized in x.  Both x and y must be positive."
    d = s * x**alpha
    return phi.pdf((y - (1 - delta) * 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)
    psi_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 = phi.rvs((n, T))
    k[:, 0] = psi_0.rvs(n)
    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 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 psi, g in zip(laes, greys):
        ax.plot(ygrid, psi(ygrid), color=g, lw=2, alpha=0.6)
    ax.set_xlabel('capital')
plt.show()
```

Exercise 3

Here's a possible solution.

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

```
n = 20
k = 5000
J = 6

theta = 0.9
d = np.sqrt(1 - theta**2)
delta = theta / 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)
    title = 'time series from t = ' + str(initial_conditions[j])
    axes[j].set_title(title)
```

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

plt.show()
```

time series from t = 8.0
time series from t = 6.4
time series from t = 4.8
time series from t = 3.2
time series from t = 1.6

Appendix

Here's the proof of (4.52)

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 (4.52)

A First Look at the Kalman Filter

Contents

- *A First Look at the Kalman Filter*
  - *Overview*
  - *The Basic Idea*
  - *Convergence*
  - *Implementation*
  - *Exercises*
  - *Solutions*

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

- *[LS12]*, 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.

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-longitute 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

$$p = N(\hat{x}, \Sigma) \tag{4.67}$$

where $\hat{x}$ is the mean of the distribution and Σ is a 2×2 covariance matrix. In our simulations, we will suppose that

$$\hat{x} = \begin{pmatrix} 0.2 \\ -0.2 \end{pmatrix}, \qquad \Sigma = \begin{pmatrix} 0.4 & 0.3 \\ 0.3 & 0.45 \end{pmatrix} \tag{4.68}$$

This density $p(x)$ is shown below as a contour map, with the center of the red ellipse being equal to $\hat{x}$

Fig. 4.1: Prior density (Click this or any other figure to enlarge.)

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

$$y = Gx + v, \quad \text{where} \quad v \sim N(0, R) \tag{4.69}$$

Here G and R are 2×2 matrices with R positive definite. Both are assumed known, and the noise term v is assumed to be independent of x

How then should we combine our prior $p(x) = N(\hat{x}, \Sigma)$ and this new information y to improve our understanding of the location of the missile?

As you may have guessed, the answer is to use Bayes' theorem, which tells us to update our prior $p(x)$ to $p(x \,|\, y)$ via

$$p(x \,|\, y) = \frac{p(y \,|\, x)\, p(x)}{p(y)}$$

where $p(y) = \int p(y \,|\, x)\, p(x) dx$

In solving for $p(x \,|\, y)$, we observe that

- $p(x) = N(\hat{x}, \Sigma)$
- In view of (4.69), 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

$$p(x \mid y) = N(\hat{x}^F, \Sigma^F)$$

where

$$\hat{x}^F := \hat{x} + \Sigma G'(G\Sigma G' + R)^{-1}(y - G\hat{x}) \quad \text{and} \quad \Sigma^F := \Sigma - \Sigma G'(G\Sigma G' + R)^{-1}G\Sigma \tag{4.70}$$

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 $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ is shown in the next figure via contour lines and the color map

The original density is left in as contour lines for comparison

Our new density twists the prior $p(x)$ in a direction determined by the new information $y - G\hat{x}$

In generating the figure, we set G to the identity matrix and $R = 0.5\Sigma$ for Σ defined in (4.68)

(The code for generating this and the proceding figures can be found in the file gaussian_contours.py

[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.

The Forecast Step

What have we achieved so far?

We have obtained probabilities for the current location of the state (missile) given prior and current information

This is called "filtering" rather than forecasting, because we are filtering out noise rather than looking into the future

- $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ is called the *filtering distribution*

But now let's suppose that we are given another task: to predict the location of the missile after one unit of time (whatever that may be) has elapsed

To do this we need a model of how the state evolves

Let's suppose that we have one, and that it's linear and Gaussian. In particular,

$$x_{t+1} = Ax_t + w_{t+1}, \quad \text{where} \quad w_t \sim N(0, Q) \tag{4.71}$$

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 (4.71), 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 (4.70) tell us that

$$\mathbb{E}[Ax^F + w] = A\mathbb{E}x^F + \mathbb{E}w = A\hat{x}^F = A\hat{x} + A\Sigma G'(G\Sigma G' + R)^{-1}(y - G\hat{x})$$

and

$$\operatorname{Var}[Ax^F + w] = A \operatorname{Var}[x^F]A' + Q = A\Sigma^F A' + Q = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q$$

The matrix $A\Sigma G'(G\Sigma G' + R)^{-1}$ is often written as K_Σ and called the *Kalman gain*

- The subscript Σ has been added to remind us that K_Σ depends on Σ, but not y or $\hat{x}$

Using this notation, we can summarize our results as follows

Our updated prediction is the density $N(\hat{x}_{new}, \Sigma_{new})$ where

$$\begin{aligned} \hat{x}_{new} &:= A\hat{x} + K_\Sigma(y - G\hat{x}) \\ \Sigma_{new} &:= A\Sigma A' - K_\Sigma G\Sigma A' + Q \end{aligned}$$

- The density $p_{new}(x) = N(\hat{x}_{new}, \Sigma_{new})$ is called the *predictive distribution*

The predictive distribution is the new density shown in the following figure, where the update has used parameters

$$A = \begin{pmatrix} 1.2 & 0.0 \\ 0.0 & -0.2 \end{pmatrix}, \qquad Q = 0.3 * \Sigma$$

The Recursive Procedure

Let's look back at what we've done

We started the current period with a prior $p(x)$ for the location x of the missile

We then used the current measurement y to update to $p(x \mid y)$

Finally, we used the law of motion (4.71) 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 $p_t(x \mid y) = N(\hat{x}_t^F, \Sigma_t^F)$ from $p_t(x)$ and y_t, applying Bayes rule and the conditional distribution (4.69)
4. Compute the predictive distribution $p_{t+1}(x) = N(\hat{x}_{t+1}, \Sigma_{t+1})$ from the filtering distribution and (4.71)
5. Increment t by one and go to step 1

Repeating (4.72), the dynamics for $\hat{x}_t$ and Σ_t are as follows

$$\begin{aligned}\hat{x}_{t+1} &= A\hat{x}_t + K_{\Sigma_t}(y_t - G\hat{x}_t) \\ \Sigma_{t+1} &= A\Sigma_t A' - K_{\Sigma_t} G\Sigma_t A' + Q\end{aligned}$$

These are the standard dynamic equations for the Kalman filter (see, for example, *[LS12]*, page 58)

Convergence

The matrix Σ_t is a measure of the uncertainty of our prediction $\hat{x}_t$ of x_t

Apart from special cases, this uncertainty will never be fully resolved, regardless of how much time elapses

One reason is that our prediction $\hat{x}_t$ is made based on information available at $t-1$, not t

Even if we know the precise value of x_{t-1} (which we don't), the transition equation (4.71) 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 Σ_t converges to a constant matrix as $t \to \infty$

To study this topic, let's expand the second equation in (4.72):

$$\Sigma_{t+1} = A\Sigma_t A' - A\Sigma_t G'(G\Sigma_t G' + R)^{-1} G\Sigma_t A' + Q \tag{4.72}$$

This is a nonlinear difference equation in Σ_t

A fixed point of (4.72) is a constant matrix Σ such that

$$\Sigma = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1} G\Sigma A' + Q \tag{4.73}$$

Equation (4.72) is known as a discrete time Riccati difference equation

Equation (4.73) 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 λ_i of A satisfy $|\lambda_i| < 1$ (cf. e.g., *[AM05]*, p. 77)

(This strong condition assures that the unconditional distribution of x_t converges as $t \to +\infty$)

In this case, for any initial choice of Σ_0 that is both nonnegative and symmetric, the sequence $\{\Sigma_t\}$ in (4.72) converges to a nonnegative symmetric matrix Σ that solves (4.73)

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 (4.73) and the corresponding (stationary) Kalman gain

You can view the program on GitHub

Exercises

Exercise 1

Consider the following simple application of the Kalman filter, loosely based on *[LS12]*, 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 (4.71) 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 *[LS12]*, sections 2.9.1–2.9.2, these distributions asymptotically put all mass on the unknown value θ

In the simulation, take $\theta = 10$, $\hat{x}_0 = 8$ and $\Sigma_0 = 1$

Your figure should – modulo randomness – look something like this

Exercise 2

The preceding figure gives some support to the idea that probability mass converges to θ

To get a better idea, choose a small $\epsilon > 0$ and calculate

$$z_t := 1 - \int_{\theta-\epsilon}^{\theta+\epsilon} p_t(x)dx$$

for $t = 0, 1, 2, \ldots, T$

Plot z_t against T, setting $\epsilon = 0.1$ and $T = 600$

Your figure should show error erratically declining something like this

Exercise 3

As discussed *above*, if the shock sequence $\{w_t\}$ is not degenerate, then it is not in general possible to predict x_t without error at time $t-1$ (and this would be the case even if we could observe x_{t-1})

Let's now compare the prediction $\hat{x}_t$ made by the Kalman filter against a competitor who **is** allowed to observe x_{t-1}

This competitor will use the conditional expectation $\mathbb{E}[x_t \,|\, x_{t-1}]$, which in this case is Ax_{t-1}

The conditional expectation is known to be the optimal prediction method in terms of minimizing mean squared error

(More precisely, the minimizer of $\mathbb{E}\,\|x_t - g(x_{t-1})\|^2$ with respect to g is $g^*(x_{t-1}) := \mathbb{E}[x_t \,|\, x_{t-1}]$)

Thus we are comparing the Kalman filter against a competitor who has more information (in the sense of being able to observe the latent state) and behaves optimally in terms of minimizing squared error

Our horse race will be assessed in terms of squared error

In particular, your task is to generate a graph plotting observations of both $\|x_t - Ax_{t-1}\|^2$ and $\|x_t - \hat{x}_t\|^2$ against t for $t = 1, \ldots, 50$

For the parameters, set $G = I$, $R = 0.5I$ and $Q = 0.3I$, where I is the 2×2 identity

Set

$$A = \begin{pmatrix} 0.5 & 0.4 \\ 0.6 & 0.3 \end{pmatrix}$$

To initialize the prior density, set

$$\Sigma_0 = \begin{pmatrix} 0.9 & 0.3 \\ 0.3 & 0.9 \end{pmatrix}$$

and $\hat{x}_0 = (8, 8)$

Finally, set $x_0 = (0, 0)$

You should end up with a figure similar to the following (modulo randomness)

Observe how, after an initial learning period, the Kalman filter performs quite well, even relative to the competitor who predicts optimally with knowledge of the latent state

Exercise 4

Try varying the coefficient 0.3 in $Q = 0.3I$ up and down

Observe how the diagonal values in the stationary solution Σ (see (4.73)) 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

Solutions

```
import numpy as np
import matplotlib.pyplot as plt
from quantecon import Kalman
from quantecon import LinearStateSpace
from scipy.stats import norm
```

Exercise 1

```
# == parameters == #
theta = 10  # Constant value of state x_t
A, C, G, H = 1, 0, 1, 1
ss = LinearStateSpace(A, C, G, H, mu_0=theta)

# == set prior, initialize kalman filter == #
x_hat_0, Sigma_0 = 8, 1
kalman = Kalman(ss, x_hat_0, Sigma_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(theta - 5, theta + 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=r'$t=%d$'
↪% i)
    kalman.update(y[i])

ax.set_title(r'First %d densities when $\theta = %.1f$' % (N, theta))
ax.legend(loc='upper left')
plt.show()
```

Exercise 2

```
from scipy.integrate import quad

epsilon = 0.1
theta = 10  # Constant value of state x_t
A, C, G, H = 1, 0, 1, 1
ss = LinearStateSpace(A, C, G, H, mu_0=theta)

x_hat_0, Sigma_0 = 8, 1
kalman = Kalman(ss, x_hat_0, Sigma_0)

T = 600
z = np.empty(T)
x, y = ss.simulate(T)
y = y.flatten()

for t in range(T):
    # Record the current predicted mean and variance, and plot their densities
```

```
    m, v = [float(temp) for temp in (kalman.x_hat, kalman.Sigma)]

    f = lambda x: norm.pdf(x, loc=m, scale=np.sqrt(v))
    integral, error = quad(f, theta - epsilon, theta + epsilon)
    z[t] = 1 - integral

    kalman.update(y[t])

fig, ax = plt.subplots(figsize=(9, 7))
ax.set_ylim(0, 1)
ax.set_xlim(0, T)
ax.plot(range(T), z)
ax.fill_between(range(T), np.zeros(T), z, color="blue", alpha=0.2)
plt.show()
```

Exercise 3

```
from numpy.random import multivariate_normal
from scipy.linalg import eigvals
```

```
# === 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 Sigma == #
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] - np.dot(A, x[:,t-1]))**2)

fig, ax = plt.subplots(figsize=(9,6))
ax.plot(range(1, T), e1, 'k-', lw=2, alpha=0.6, label='Kalman filter error')
ax.plot(range(1, T), e2, 'g-', lw=2, alpha=0.6, label='conditional␣
↪expectation error')
ax.legend()
plt.show()
```

```
Eigenvalues of A:
[ 0.9+0.j -0.1+0.j]
Stationary prediction error variance:
[[ 0.40329108  0.1050718 ]
 [ 0.1050718   0.41061709]]
```

LQ Dynamic Programming Problems

Contents

- *LQ Dynamic Programming Problems*
  - *Overview*
  - *Introduction*
  - *Optimality – Finite Horizon*
  - *Extensions and Comments*
  - *Implementation*
  - *Further Applications*
  - *Exercises*
  - *Solutions*

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,

- *[LS12]*, 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)

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

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \qquad t = 0, 1, 2, \ldots \tag{4.74}$$

Here

- u_t is a "control" vector, incorporating choices available to a decision maker confronting the current state x_t
- $\{w_t\}$ is an uncorrelated zero mean shock process satisfying $\mathbb{E}w_t w_t' = I$, where the right-hand side is the identity matrix

Regarding the dimensions

- x_t is $n \times 1$, A is $n \times n$
- u_t is $k \times 1$, B is $n \times k$
- w_t is $j \times 1$, C is $n \times j$

Example 1

Consider a household budget constraint given by

$$a_{t+1} + c_t = (1 + r)a_t + y_t$$

Here a_t is assets, r is a fixed interest rate, c_t is current consumption, and y_t is current non-financial income

If we suppose that $\{y_t\}$ is serially uncorrelated and $N(0, \sigma^2)$, then, taking $\{w_t\}$ to be standard normal, we can write the system as

$$a_{t+1} = (1 + r)a_t - c_t + \sigma w_{t+1}$$

This is clearly a special case of (4.74), with assets being the state and consumption being the control

Example 2

One unrealistic feature of the previous model is that non-financial income has a zero mean and is often negative

This can easily be overcome by adding a sufficiently large mean

Hence in this example we take $y_t = \sigma w_{t+1} + \mu$ for some positive real number μ

Another alteration that's useful to introduce (we'll see why soon) is to change the control variable from consumption to the deviation of consumption from some "ideal" quantity $\bar{c}$

(Most parameterizations will be such that $\bar{c}$ is large relative to the amount of consumption that is attainable in each period, and hence the household wants to increase consumption)

For this reason, we now take our control to be $u_t := c_t - \bar{c}$

In terms of these variables, the budget constraint $a_{t+1} = (1 + r)a_t - c_t + y_t$ becomes

$$a_{t+1} = (1 + r)a_t - u_t - \bar{c} + \sigma w_{t+1} + \mu \tag{4.75}$$

How can we write this new system in the form of equation (4.74)?

If, as in the previous example, we take a_t as the state, then we run into a problem: the law of motion contains some constant terms on the right-hand side

This means that we are dealing with an *affine* function, not a linear one (recall *this discussion*)

Fortunately, we can easily circumvent this problem by adding an extra state variable

In particular, if we write

$$\begin{pmatrix} a_{t+1} \\ 1 \end{pmatrix} = \begin{pmatrix} 1+r & -\bar{c}+\mu \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_t \\ 1 \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix} u_t + \begin{pmatrix} \sigma \\ 0 \end{pmatrix} w_{t+1} \tag{4.76}$$

then the first row is equivalent to (4.75)

Moreover, the model is now linear, and can be written in the form of (4.74) by setting

$$x_t := \begin{pmatrix} a_t \\ 1 \end{pmatrix}, \quad A := \begin{pmatrix} 1+r & -\bar{c}+\mu \\ 0 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \end{pmatrix} \tag{4.77}$$

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

$$x_t' R x_t + u_t' Q u_t \tag{4.78}$$

Here

- R is assumed to be $n \times n$, symmetric and nonnegative definite
- Q is assumed to be $k \times k$, symmetric and positive definite

---

Note: In fact, for many economic problems, the definiteness conditions on R and Q can be relaxed. It is sufficient that certain submatrices of R and Q be nonnegative definite. See *[HS08]* for details

---

Example 1

A very simple example that satisfies these assumptions is to take R and Q to be identity matrices, so that current loss is

$$x_t' I x_t + u_t' I u_t = \|x_t\|^2 + \|u_t\|^2$$

Thus, for both the state and the control, loss is measured as squared distance from the origin

(In fact the general case (4.78) can also be understood in this way, but with R and Q identifying other – non-Euclidean – notions of "distance" from the zero vector)

Intuitively, we can often think of the state x_t as representing deviation from a target, such as

- deviation of inflation from some target level
- deviation of a firm's capital stock from some desired quantity

The aim is to put the state close to the target, while using controls parsimoniously

Example 2

In the household problem *studied above*, setting $R = 0$ and $Q = 1$ yields preferences

$$x_t' R x_t + u_t' Q u_t = u_t^2 = (c_t - \bar{c})^2$$

Under this specification, the household's current loss is the squared deviation of consumption from the ideal level $\bar{c}$

Optimality – Finite Horizon

Let's 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

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (x_t' R x_t + u_t' Q u_t) + \beta^T x_T' R_f x_T \right\} \tag{4.79}$$

subject to the law of motion (4.74) and initial state x_0

The new objects introduced here are β and the matrix R_f

The scalar β is the discount factor, while $x' R_f x$ gives terminal loss associated with state x

Comments:

- We assume R_f to be $n \times n$, symmetric and nonnegative definite
- We allow $\beta = 1$, and hence include the undiscounted case
- x_0 may itself be random, in which case we require it to be independent of the shock sequence $w_1, \ldots, w_T$

Information

There's one constraint we've neglected to mention so far, which is that the decision maker who solves this LQ problem knows only the present and the past, not the future

To clarify this point, consider the sequence of controls $\{u_0, \ldots, u_{T-1}\}$

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 σ-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

$$\min_u \{x'_{T-1} R x_{T-1} + u' Q u + \beta \, \mathbb{E} J_T(A x_{T-1} + B u + C w_T)\}$$

At this stage, it is convenient to define the function

$$J_{T-1}(x) = \min_u \{x' R x + u' Q u + \beta \, \mathbb{E} J_T(A x + B u + C w_T)\} \tag{4.80}$$

The function J_{T-1} will be called the $T-1$ value function, and $J_{T-1}(x)$ can be thought of as representing total "loss-to-go" from state x at time $T-1$ when the decision maker behaves optimally

Now let's step back to $T-2$

For a decision maker at $T-2$, the value $J_{T-1}(x)$ plays a role analogous to that played by the terminal loss $J_T(x) = x' R_f x$ for the decision maker at $T-1$

That is, $J_{T-1}(x)$ summarizes the future loss associated with moving to state x

The decision maker chooses her control u to trade off current loss against future loss, where

- the next period state is $x_{T-1} = Ax_{T-2} + Bu + Cw_{T-1}$, 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

$$\min_u \{x'_{T-2} R x_{T-2} + u'Qu + \beta \, \mathbb{E} J_{T-1}(Ax_{T-2} + Bu + Cw_{T-1})\}$$

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

$$J_{t-1}(x) = \min_u \{x'Rx + u'Qu + \beta \, \mathbb{E} J_t(Ax + Bu + Cw_t)\} \quad \text{and} \quad J_T(x) = x'R_f x$$

The first equality is the Bellman equation from dynamic programming theory specialized to the finite horizon LQ problem

Now that we have $\{J_0, \ldots, J_T\}$, we can obtain the optimal controls

As a first step, let's find out what the value functions look like

It turns out that every J_t has the form $J_t(x) = x'P_t x + d_t$ where P_t is a $n \times n$ matrix and d_t is a constant

We can show this by induction, starting from $P_T := R_f$ and $d_T = 0$

Using this notation, (4.80) becomes

$$J_{T-1}(x) = \min_u \{x'Rx + u'Qu + \beta \, \mathbb{E}(Ax + Bu + Cw_T)' P_T (Ax + Bu + Cw_T)\} \tag{4.81}$$

To obtain the minimizer, we can take the derivative of the r.h.s. with respect to u and set it equal to zero

Applying the relevant rules of *matrix calculus*, this gives

$$u = -(Q + \beta B' P_T B)^{-1} \beta B' P_T A x \tag{4.82}$$

Plugging this back into (4.81) and rearranging yields

$$J_{T-1}(x) = x' P_{T-1} x + d_{T-1}$$

where

$$P_{T-1} = R - \beta^2 A' P_T B (Q + \beta B' P_T B)^{-1} B' P_T A + \beta A' P_T A \tag{4.83}$$

and

$$d_{T-1} := \beta \, \mathrm{trace}(C' P_T C) \tag{4.84}$$

(The algebra is a good exercise — we'll leave it up to you)

If we continue working backwards in this manner, it soon becomes clear that $J_t(x) = x'P_t x + d_t$ as claimed, where $\{P_t\}$ and $\{d_t\}$ satisfy the recursions

$$P_{t-1} = R - \beta^2 A'P_t B(Q + \beta B'P_t B)^{-1} B'P_t A + \beta A'P_t A \quad \text{with} \quad P_T = R_f \tag{4.85}$$

and

$$d_{t-1} = \beta(d_t + \text{trace}(C'P_t C)) \quad \text{with} \quad d_T = 0 \tag{4.86}$$

Recalling (4.82), the minimizers from these backward steps are

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B'P_{t+1}B)^{-1}\beta B'P_{t+1}A \tag{4.87}$$

These are the linear optimal control policies we *discussed above*

In particular, the sequence of controls given by (4.87) and (4.74) solves our finite horizon LQ problem

Rephrasing this more precisely, the sequence $u_0, \ldots, u_{T-1}$ given by

$$u_t = -F_t x_t \quad \text{with} \quad x_{t+1} = (A - BF_t)x_t + Cw_{t+1} \tag{4.88}$$

for $t = 0, \ldots, T-1$ attains the minimum of (4.79) subject to our constraints

An Application

Early Keynesian models assumed that households have a constant marginal propensity to consume from current income

Data contradicted the constancy of the marginal propensity to consume

In response, Milton Friedman, Franco Modigliani and others built models based on a consumer's preference for an intertemporally smooth consumption stream

(See, for example, *[Fri56]* or *[MB54]*)

One property of those models is that households purchase and sell financial assets to make consumption streams smoother than income streams

The household savings problem *outlined above* captures these ideas

The optimization problem for the household is to choose a consumption sequence in order to minimize

$$\mathbb{E}\left\{\sum_{t=0}^{T-1} \beta^t (c_t - \bar{c})^2 + \beta^T q a_T^2\right\} \tag{4.89}$$

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 (4.75)

We saw how this constraint could be manipulated into the LQ formulation $x_{t+1} = Ax_t + Bu_t + Cw_{t+1}$ by setting $x_t = (a_t \ 1)'$ and using the definitions in (4.77)

To match with this state and control, the objective function (4.89) can be written in the form of (4.79) 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 (4.85) and (4.87)

After generating shocks $w_1, \ldots, w_T$, the dynamics for assets and consumption can be simulated via (4.88)

We provide code for all these operations below

The following figure was computed using this code, with $r = 0.05, \beta = 1/(1+r), \bar{c} = 2, \mu = 1, \sigma = 0.25, T = 45$ and $q = 10^6$

The shocks $\{w_t\}$ were taken to be iid and standard normal

The top panel shows the time path of consumption c_t and income y_t in the simulation

As anticipated by the discussion on consumption smoothing, the time path of consumption is much smoother than that for income

(But note that consumption becomes more irregular towards the end of life, when the zero final asset requirement impinges more on consumption choices)

The second panel in the figure shows that the time path of assets a_t is closely correlated with cumulative unanticipated income, where the latter is defined as

$$z_t := \sum_{j=0}^{t} \sigma w_t$$

A key message is that unanticipated windfall gains are saved rather than consumed, while unanticipated negative shocks are met by reducing assets

(Again, this relationship breaks down towards the end of life due to the zero final asset requirement)

These results are relatively robust to changes in parameters

For example, let's increase β from $1/(1+r) \approx 0.952$ to 0.96 while keeping other parameters fixed

This consumer is slightly more patient than the last one, and hence puts relatively more weight on later consumption values

A simulation is shown below

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

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

$$\mathbb{E}\left\{\sum_{t=0}^{T-1} \beta^t (x_t' R x_t + u_t' Q u_t + 2 u_t' N x_t) + \beta^T x_T' R_f x_T\right\} \tag{4.90}$$

Our results extend to this case in a straightforward way

The sequence $\{P_t\}$ from (4.85) becomes

$$P_{t-1} = R - (\beta B' P_t A + N)'(Q + \beta B' P_t B)^{-1}(\beta B' P_t A + N) + \beta A' P_t A \quad \text{with} \quad P_T = R_f \tag{4.91}$$

The policies in (4.87) are modified to

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B' P_{t+1} B)^{-1}(\beta B' P_{t+1} A + N) \tag{4.92}$$

The sequence $\{d_t\}$ is unchanged from (4.86)

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

$$\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t (x_t' R x_t + u_t' Q u_t + 2 u_t' N x_t)\right\} \tag{4.93}$$

In the infinite horizon case, optimal policies can depend on time only if time itself is a component of the state vector x_t

In other words, there exists a fixed matrix F such that $u_t = -F x_t$ for all t

That decision rules are constant over time is intuitive — after all, the decision maker faces the same infinite horizon at every stage, with only the current state changing

Not surprisingly, P and d are also constant

The stationary matrix P is the solution to the discrete time algebraic Riccati equation

$$P = R - (\beta B' P A + N)'(Q + \beta B' P B)^{-1}(\beta B' P A + N) + \beta A' P A \tag{4.94}$$

Equation (4.94) is also called the *LQ Bellman equation*, and the map that sends a given P into the right-hand side of (4.94) is called the *LQ Bellman operator*

The stationary optimal policy for this model is

$$u = -Fx \quad \text{where} \quad F = (Q + \beta B'PB)^{-1}(\beta B'PA + N) \tag{4.95}$$

The sequence $\{d_t\}$ from (4.86) is replaced by the constant value

$$d := \text{trace}(C'PC)\frac{\beta}{1-\beta} \tag{4.96}$$

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 (4.95) or (4.92)

It follows that we can ignore uncertainty when solving for optimal behavior, and plug it back in when examining optimal state dynamics

Implementation

We have put together some code for solving finite and infinite horizon linear quadratic control problems

The code can be found in the file `lqcontrol.py` from the QuantEcon.py package

You can view the program on GitHub

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 = 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 (4.85), (4.86) and (4.87)
  - `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 example of usage is given in lq_permanent_1.py, the contents of which are shown below

This program can be used to replicate the figures shown in our *section on the permanent income model*

(Some of the plotting techniques are rather fancy and you can ignore those details if you wish)

```
"""
Filename: lq_permanent_1.py
Authors: John Stachurski and Thomas J. Sargent

A permanent income / life-cycle model with iid income
"""

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)

# == Convert back to assets, consumption and income == #
assets = xp[0, :]            # a_t
c = up.flatten() + c_bar     # c_t
income = wp[0, 1:] + mu      # y_t

# == Plot results == #
n_rows = 2
fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))
```

```
plt.subplots_adjust(hspace=0.5)
for i in range(n_rows):
    axes[i].grid()
    axes[i].set_xlabel(r'Time')
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[0].legend(ncol=2, **legend_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-')
axes[1].legend(ncol=2, **legend_args)

plt.show()
```

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 end of working life, and falls more during retirement

In this section, we will model this rise and fall as a symmetric inverted "U" using a polynomial in age

As before, the consumer seeks to minimize

$$\mathbb{E}\left\{\sum_{t=0}^{T-1}\beta^t(c_t-\bar{c})^2+\beta^T q a_T^2\right\} \tag{4.97}$$

subject to $a_{t+1}=(1+r)a_t-c_t+y_t,\ t\geq 0$

For income we now take $y_t=p(t)+\sigma w_{t+1}$ where $p(t):=m_0+m_1t+m_2t^2$

(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

$$a_{t+1} = (1+r)a_t - u_t - \bar{c} + m_1 t + m_2 t^2 + \sigma w_{t+1} \tag{4.98}$$

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

$$x_t := \begin{pmatrix} a_t \\ 1 \\ t \\ t^2 \end{pmatrix}, \quad A := \begin{pmatrix} 1+r & -\bar{c} & m_1 & m_2 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{4.99}$$

If you expand the expression $x_{t+1} = Ax_t + Bu_t + Cw_{t+1}$ using this specification, you will find that assets follow (4.98) as desired, and that the other state variables also update appropriately

To implement preference specification (4.97) we take

$$Q := 1, \quad R := \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad R_f := \begin{pmatrix} q & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{4.100}$$

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

$$y_t = \begin{cases} p(t) + \sigma w_{t+1} & \text{if } t \leq K \\ s & \text{otherwise} \end{cases} \tag{4.101}$$

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 (4.89)

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 (4.101) at K makes it very difficult to express the law of motion as a fixed-coefficient linear system

However, we can still use our LQ methods here by suitably linking two component LQ problems

These two LQ problems describe the consumer's behavior during her working life (`lq_working`) and retirement (`lq_retired`)

(This is possible because in the two separate periods of life, the respective income processes [polynomial trend and constant] each fit the LQ framework)

The basic idea is that although the whole problem is not a single time-invariant LQ problem, it is still a dynamic programming problem, and hence we can use appropriate Bellman equations at every stage

Based on this logic, we can

1. solve `lq_retired` by the usual 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

$$\mathbb{E}\left\{\sum_{t=0}^{\infty}\beta^t\pi_t\right\} \quad \text{where} \quad \pi_t := p_t q_t - cq_t - \gamma(q_{t+1} - q_t)^2 \tag{4.102}$$

Here

- $\gamma(q_{t+1} - q_t)^2$ represents adjustment costs
- c is average cost of production

This can be formulated as an LQ problem and then solved and simulated, but first let's study the problem and try to get some intuition

One way to start thinking about the problem is to consider what would happen if $\gamma = 0$

Without adjustment costs there is no intertemporal trade-off, so the monopolist will choose output to maximize current profit in each period

It's not difficult to show that profit-maximizing output is

$$\bar{q}_t := \frac{a_0 - c + d_t}{2a_1}$$

In light of this discussion, what we might expect for general γ is that

- if γ is close to zero, then q_t will track the time path of $\bar{q}_t$ relatively closely
- if γ is larger, then q_t will be smoother than $\bar{q}_t$, as the monopolist seeks to avoid adjustment costs

This intuition turns out to be correct

The following figures show simulations produced by solving the corresponding LQ problem

The only difference in parameters across the figures is the size of γ

$\bar{q}_t$
q_t
dynamics with $\gamma = 1$
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
2.2
2.0
0
20
40
60
80
100
120
140
Time

$\bar{q}_t$
q_t
dynamics with $\gamma = 10$
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
2.2
2.0
0
20
40
60
80
100
120
140
Time

To produce these figures we converted the monopolist problem into an LQ problem

The key to this conversion is to choose the right state — which can be a bit of an art

Here we take $x_t = (\bar{q}_t \;\; q_t \;\; 1)'$, while the control is chosen as $u_t = q_{t+1} - q_t$

We also manipulated the profit function slightly

In (4.102), current profits are $\pi_t := p_t q_t - c q_t - \gamma(q_{t+1} - q_t)^2$

Let's now replace π_t in (4.102) with $\hat{\pi}_t := \pi_t - a_1 \bar{q}_t^2$

This makes no difference to the solution, since $a_1 \bar{q}_t^2$ does not depend on the controls

(In fact we are just adding a constant term to (4.102), and optimizers are not affected by constant terms)

The reason for making this substitution is that, as you will be able to verify, $\hat{\pi}_t$ reduces to the simple quadratic

$$\hat{\pi}_t = -a_1 (q_t - \bar{q}_t)^2 - \gamma u_t^2$$

After negation to convert to a minimization problem, the objective becomes

$$\min \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left\{ a_1 (q_t - \bar{q}_t)^2 + \gamma u_t^2 \right\} \tag{4.103}$$

It's now relatively straightforward to find R and Q such that (4.103) can be written as (4.93)

Furthermore, the matrices A, B and C from (4.74) 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

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 (4.99)

- 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 (4.100)

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 γ varies between 1 and 50 (see figures)

Solutions

```
import numpy as np
import matplotlib.pyplot as plt
from quantecon import LQ
```

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
- $p(T) = 0$.

```
# == 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
c = up.flatten() + c_bar    # Consumption
```

```
time = np.arange(1, T+1)
income = wp[0, 1:] + m1 * time + m2 * time**2  # Income


# == Plot results == #
n_rows = 2
fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))

plt.subplots_adjust(hspace=0.5)
for i in range(n_rows):
    axes[i].grid()
    axes[i].set_xlabel(r'Time')
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[0].legend(ncol=2, **legend_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-')
axes[1].legend(ncol=1, **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.

```
# == 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
m1      = 2 * 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 = [[sigma],
     [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, :]                # Assets

up = np.column_stack((up_w, up_r))
c = up.flatten() + c_bar     # Consumption

time = np.arange(1, K+1)
income_w = wp_w[0, 1:K+1] + m1 * time + m2 * time**2   # Income
income_r = np.ones(T-K) * s
income = np.concatenate((income_w, income_r))

# == Plot results == #
n_rows = 2
fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))

plt.subplots_adjust(hspace=0.5)
for i in range(n_rows):
    axes[i].grid()
    axes[i].set_xlabel(r'Time')
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[0].legend(ncol=2, **legend_args)

axes[1].plot(range(T+1), assets, 'b-', label="assets", **p_args)
axes[1].plot(range(T+1), np.zeros(T+1), 'k-')
axes[1].legend(ncol=1, **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 $x_t = (\bar{q}_t \;\; q_t \;\; 1)'$, while $u_t = q_{t+1} - q_t$

Letting $m_0 := (a_0 - c)/2a_1$ and $m_1 := 1/2a_1$, we can write $\bar{q}_t = m_0 + m_1 d_t$, and then, with some manipulation

$$\bar{q}_{t+1} = m_0(1-\rho) + \rho\bar{q}_t + m_1\sigma w_{t+1}$$

By our definition of u_t, the dynamics of q_t are $q_{t+1} = q_t + u_t$

Using these facts you should be able to build the correct A, B, C matrices (and then check them against those found in the solution code below)

Suitable R, Q matrices can be found by inspecting the objective function, which we repeat here for 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

```
# == 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))
ax.set_xlabel('Time')

# == Some fancy plotting stuff -- simplify if you prefer == #
bbox = (0., 1.01, 1., .101)
legend_args = {'bbox_to_anchor' : bbox, 'loc' : 3, 'mode' : 'expand'}
p_args = {'lw' : 2, 'alpha' : 0.6}
```

```
time = range(len(q))
ax.set_xlim(0, max(time))
ax.plot(time, q_bar, 'k-', lw=2, alpha=0.6, label=r'$\bar q_t$')
ax.plot(time, q, 'b-', lw=2, alpha=0.6, label=r'$q_t$')
ax.legend(ncol=2, **legend_args)
s = r'dynamics with $\gamma = {}$'.format(gamma)
ax.text(max(time) * 0.6, 1 * q_bar.max(), s, fontsize=14)
plt.show()
```

Covariance Stationary Processes

Contents

- *Covariance Stationary Processes*
  - *Overview*
  - *Introduction*
  - *Spectral Analysis*

– *Implementation*

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 advantange 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

- *[LS12]*, chapter 2
- *[Sar87]*, chapter 11
- John Cochrane's notes on time series analysis, chapter 8

- *[Shi95]*, chapter 6
- *[CC08]*, all

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 $\gamma \colon \mathbb{Z} \to \mathbb{R}$ is called the *autocovariance function* of the process

Throughout this lecture, we will work exclusively with zero-mean (i.e., $\mu = 0$) covariance stationary processes

The zero-mean assumption costs nothing in terms of generality, since working with non-zero-mean processes involves no more than adding a constant

Example 1: White Noise

Perhaps the simplest class of covariance stationary processes is the white noise processes

A process $\{\epsilon_t\}$ is called a *white noise process* if

1. $\mathbb{E}\epsilon_t = 0$
2. $\gamma(k) = \sigma^2 \mathbf{1}\{k = 0\}$ for some $\sigma > 0$

(Here $\mathbf{1}\{k = 0\}$ is defined to be 1 if $k = 0$ and zero otherwise)

White noise processes play the role of **building blocks** for processes with more complicated dynamics

Example 2: General Linear Processes

From the simple building block provided by white noise, we can construct a very flexible family of covariance stationary processes — the *general linear processes*

$$X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}, \qquad t \in \mathbb{Z} \tag{4.104}$$

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 (4.104) 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 (4.104) is

$$\gamma(k) = \sigma^2 \sum_{j=0}^{\infty} \psi_j \psi_{j+k} \tag{4.105}$$

By the Cauchy-Schwartz inequality one can show that $\gamma(k)$ satisfies equation (4.105)

Evidently, $\gamma(k)$ does not depend on t

Wold's 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
- η_t can be expressed as a linear function of $X_{t-1}, X_{t-2}, \ldots$ and is perfectly predictable over arbitrarily long horizons

For 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 autoregessive or AR(1) process

$$X_t = \phi X_{t-1} + \epsilon_t \quad \text{where} \quad |\phi| < 1 \quad \text{and } \{\epsilon_t\} \text{ is white noise} \tag{4.106}$$

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 (4.105) to the previous expression for X_t, we get the AR(1) autocovariance function

$$\gamma(k) = \phi^k \frac{\sigma^2}{1-\phi^2}, \qquad k = 0, 1, \ldots \tag{4.107}$$

The next figure plots example of this function for $\phi = 0.8$ and $\phi = -0.8$ with $\sigma = 1$

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

$$X_t = \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q} \tag{4.108}$$

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 (4.108) as

$$L^0 X_t - \phi_1 L^1 X_t - \cdots - \phi_p L^p X_t = L^0 \epsilon_t + \theta_1 L^1 \epsilon_t + \cdots + \theta_q L^q \epsilon_t \tag{4.109}$$

If we let $\phi(z)$ and $\theta(z)$ be the polynomials

$$\phi(z) := 1 - \phi_1 z - \cdots - \phi_p z^p \quad \text{and} \quad \theta(z) := 1 + \theta_1 z + \cdots + \theta_q z^q \tag{4.110}$$

then (4.109) becomes

$$\phi(L) X_t = \theta(L) \epsilon_t \tag{4.111}$$

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 convariance 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*

Spectral Analysis

Autocovariance functions provide a great deal of infomation 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 $(xu - vy, xv + yu)$, while addition is standard pointwise vector addition

When endowed with these notions of multiplication and addition, the set of complex numbers forms a field — addition and multiplication play well together, just as they do in $\mathbb{R}$

The complex number (x, y) is often written as $x + iy$, where i is called the *imaginary unit*, and is understood to obey $i^2 = -1$

The $x + iy$ notation provides an easy way to remember the definition of multiplication given above, because, proceeding naively,

$$(x + iy)(u + iv) = xu - yv + i(xv + yu)$$

Converted back to our first notation, this becomes $(xu - vy, xv + yu)$ as promised

Complex numbers can be represented in the polar form $re^{i\omega}$ where

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

where $x = r\cos(\omega), y = r\sin(\omega)$, and $\omega = \arctan(y/z)$ or $\tan(\omega) = y/x$

Spectral Densities

Let $\{X_t\}$ be a covariance stationary process with autocovariance function γ satisfying $\sum_k \gamma(k)^2 < \infty$

The *spectral density* f of $\{X_t\}$ is defined as the discrete time Fourier transform of its autocovariance function γ

$$f(\omega) := \sum_{k \in \mathbb{Z}} \gamma(k) e^{-i\omega k}, \qquad \omega \in \mathbb{R}$$

(Some authors normalize the expression on the right by constants such as $1/\pi$ — the convention chosen makes little difference provided you are consistent)

Using the fact that γ is *even*, in the sense that $\gamma(t) = \gamma(-t)$ for all t, we can show that

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k) \tag{4.112}$$

It is not difficult to confirm that f is

- real-valued
- even ($f(\omega) = f(-\omega)$), and
- 2π-periodic, in the sense that $f(2\pi + \omega) = f(\omega)$ for all ω

It follows that the values of f on $[0, \pi]$ determine the values of f on all of $\mathbb{R}$ — the proof is an exercise

For this reason it is standard to plot the spectral density only on the interval $[0, \pi]$

Example 1: White Noise

Consider a white noise process $\{\epsilon_t\}$ with standard deviation σ

It is easy to check that in this case $f(\omega) = \sigma^2$. So f is a constant function

As we will see, this can be interpreted as meaning that "all frequencies are equally present"

(White light has this property when frequency refers to the visible spectrum, a connection that provides the origins of the term "white noise")

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

$$f(\omega) = \sigma^2 (1 + 2\theta \cos(\omega) + \theta^2) \tag{4.113}$$

With a bit more effort, it's possible to show (see, e.g., p. 261 of *[Sar87]*) that the spectral density of the AR(1) process $X_t = \phi X_{t-1} + \epsilon_t$ is

$$f(\omega) = \frac{\sigma^2}{1 - 2\phi \cos(\omega) + \phi^2} \tag{4.114}$$

More generally, it can be shown that the spectral density of the ARMA process (4.108) is

$$f(\omega) = \left| \frac{\theta(e^{i\omega})}{\phi(e^{i\omega})} \right|^2 \sigma^2 \tag{4.115}$$

where

- σ is the standard deviation of the white noise process $\{\epsilon_t\}$
- the polynomials $\phi(\cdot)$ and $\theta(\cdot)$ are as defined in (4.110)

The derivation of (4.115) 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 (4.113) and (4.114) are indeed special cases of (4.115)

Interpreting the Spectral Density

Plotting (4.114) reveals the shape of the spectral density for the AR(1) model when ϕ takes the values 0.8 and -0.8 respectively

These spectral densities correspond to the autocovariance functions for the AR(1) process *shown above*

Informally, we think of the spectral density as being large at those $\omega \in [0, \pi]$ at which the autocovariance function seems approximately to exhibit big damped cycles

To see the idea, let's consider why, in the lower panel of the preceding figure, the spectral density for the case $\phi = -0.8$ is large at $\omega = \pi$

Recall that the spectral density can be expressed as

$$f(\omega) = \gamma(0) + 2\sum_{k\geq 1}\gamma(k)\cos(\omega k) = \gamma(0) + 2\sum_{k\geq 1}(-0.8)^k\cos(\omega k) \tag{4.116}$$

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 (4.116) is large

These ideas are illustrated in the next figure, which has k on the horizontal axis (click to enlarge)

On the other hand, if we evaluate $f(\omega)$ at $\omega = \pi/3$, then the cycles are not matched, the sequence $\gamma(k)\cos(\omega k)$ contains both positive and negative terms, and hence the sum of these terms is much smaller

In summary, the spectral density is large at frequencies ω 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

$$\gamma(k) = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(\omega) e^{i\omega k} d\omega \tag{4.117}$$

This is convenient in situations where the spectral density is easier to calculate and manipulate than the autocovariance function

(For example, the expression (4.115) 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

$$f = \sum_k \alpha_k h_k \quad \text{where} \quad \alpha_k := \langle f, h_k \rangle \tag{4.118}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product in H

Thus, f can be represented to any degree of precision by linearly combining basis vectors

The scalar sequence $\alpha = \{\alpha_k\}$ is called the *Fourier coefficients* of f, and satisfies $\sum_k |\alpha_k|^2 < \infty$

In other words, α is in ℓ_2, the set of square summable sequences

Consider an operator T that maps $\alpha \in \ell_2$ into its expansion $\sum_k \alpha_k h_k \in H$

The Fourier coefficients of $T\alpha$ are just $\alpha = \{\alpha_k\}$, as you can verify by confirming that $\langle T\alpha, h_k \rangle = \alpha_k$

Using elementary results from Hilbert space theory, it can be shown that

- T is one-to-one — if α and β are distinct in ℓ_2, then so are their expansions in H
- T is onto — if $f \in H$ then its preimage in ℓ_2 is the sequence α given by $\alpha_k = \langle f, h_k \rangle$
- T is a linear isometry — in particular $\langle \alpha, \beta \rangle = \langle T\alpha, T\beta \rangle$

Summarizing these results, we say that any separable Hilbert space is isometrically isomorphic to ℓ_2

In essence, this says that each separable Hilbert space we consider is just a different way of looking at the fundamental space ℓ_2

With this in mind, let's specialize to a setting where

- $\gamma \in \ell_2$ is the autocovariance function of a covariance stationary process, and f is the spectral density
- $H = L_2$, where L_2 is the set of square summable functions on the interval $[-\pi, \pi]$, with inner product $\langle g, h \rangle = \int_{-\pi}^{\pi} g(\omega) h(\omega) d\omega$
- $\{h_k\} =$ the orthonormal basis for L_2 given by the set of trigonometric functions

$$h_k(\omega) = \frac{e^{i\omega k}}{\sqrt{2\pi}}, \quad k \in \mathbb{Z}, \quad \omega \in [-\pi, \pi]$$

Using the definition of T from above and the fact that f is even, we now have

$$T\gamma = \sum_{k \in \mathbb{Z}} \gamma(k) \frac{e^{i\omega k}}{\sqrt{2\pi}} = \frac{1}{\sqrt{2\pi}} f(\omega) \tag{4.119}$$

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 γ

In particular, it is an expansion of the autocovariance function with respect to the trigonometric basis functions in L_2

As discussed above, the Fourier coefficients of $T\gamma$ are given by the sequence γ, and, in particular, $\gamma(k) = \langle T\gamma, h_k \rangle$

Transforming this inner product into its integral expression and using (4.119) gives (4.117), justifying our earlier expression for the inverse transform

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

In additional to individual plots of these entities, we provide functionality to generate 2x2 plots containing all this information

In other words, we want to replicate the plots on pages 68–69 of *[LS12]*

Here's an example corresponding to the model $X_t = 0.5X_{t-1} + \epsilon_t - 0.8\epsilon_{t-2}$

Code

You can view the code on GitHub

Here's an example of usage

```
import quantecon as qe

phi = 0.5
theta = 0, -0.8
lp = qe.ARMA(phi, theta)
lp.quad_plot()
```

Explanation

The call

```
lp = ARMA(phi, theta, sigma)
```

creates an instance `lp` that represents the ARMA(p, q) model

$$X_t = \phi_1 X_{t-1} + ... + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + ... + \theta_q \epsilon_{t-q}$$

If `phi` and `theta` are arrays or sequences, then the interpretation will be

- `phi` holds the vector of parameters $(\phi_1, \phi_2, ..., \phi_p)$
- `theta` holds the vector of parameters $(\theta_1, \theta_2, ..., \theta_q)$

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 (4.111)

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 = dimpulse((ma_poly, ar_poly, 1), n=impulse_length)
```

where `ma_poly` and `ar_poly` correspond to the polynomials in (4.110) — 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 `phi` or `theta` ex-post by assignments such as `lp.phi = (0.5, 0.2)` or `lp.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 NumPy's Fourier transform package *np.fft*, which wraps a standard Fortran-based package called FFTPACK

A look at the np.fft documentation shows that the inverse transform *np.fft.ifft* takes a given sequence $A_0, A_1, \ldots, A_{n-1}$ and returns the sequence $a_0, a_1, \ldots, a_{n-1}$ defined by

$$a_k = \frac{1}{n} \sum_{t=0}^{n-1} A_t e^{ik2\pi t/n}$$

Thus, if we set $A_t = f(\omega_t)$, where f is the spectral density and $\omega_t := 2\pi t/n$, then

$$a_k = \frac{1}{n}\sum_{t=0}^{n-1} f(\omega_t)e^{i\omega_t k} = \frac{1}{2\pi}\frac{2\pi}{n}\sum_{t=0}^{n-1} f(\omega_t)e^{i\omega_t k}, \qquad \omega_t := 2\pi t/n$$

For n sufficiently large, we then have

$$a_k \approx \frac{1}{2\pi}\int_0^{2\pi} f(\omega)e^{i\omega k}d\omega = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(\omega)e^{i\omega k}d\omega$$

(You can check the last equality)

In view of (4.117) we have now shown that, for n sufficiently large, $a_k \approx \gamma(k)$ — which is exactly what we want to compute

Estimation of Spectra

Contents

- *Estimation of Spectra*
  - *Overview*
  - *Periodograms*
  - *Smoothing*
  - *Exercises*
  - *Solutions*

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]*.

Periodograms

Recall that the spectral density f of a covariance stationary process with autocorrelation function γ can be written

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k), \qquad \omega \in \mathbb{R}$$

Now consider the problem of estimating the spectral density of a given time series, when γ is unknown

In particular, let $X_0, \ldots, X_{n-1}$ be n consecutive observations of a single time series that is assumed to be covariance stationary

The most common estimator of the spectral density of this process is the *periodogram* of $X_0, \ldots, X_{n-1}$, which is defined as

$$I(\omega) := \frac{1}{n} \left| \sum_{t=0}^{n-1} X_t e^{it\omega} \right|^2, \qquad \omega \in \mathbb{R} \tag{4.120}$$

(Recall that $|z|$ denotes the modulus of complex number z)

Alternatively, $I(\omega)$ can be expressed as

$$I(\omega) = \frac{1}{n} \left\{ \left[\sum_{t=0}^{n-1} X_t \cos(\omega t) \right]^2 + \left[\sum_{t=0}^{n-1} X_t \sin(\omega t) \right]^2 \right\}$$

It is straightforward to show that the function I is even and 2π-periodic (i.e., $I(\omega) = I(-\omega)$ and $I(\omega + 2\pi) = I(\omega)$ for all $\omega \in \mathbb{R}$)

From these two results, you will be able to verify that the values of I on $[0, \pi]$ determine the values of I on all of $\mathbb{R}$

The next section helps to explain the connection between the periodogram and the spectral density

Interpretation

To interpret the periodogram, it is convenient to focus on its values at the *Fourier frequencies*

$$\omega_j := \frac{2\pi j}{n}, \quad j = 0, \ldots, n-1$$

In what sense is $I(\omega_j)$ an estimate of $f(\omega_j)$?

The answer is straightforward, although it does involve some algebra

With a bit of effort one can show that, for any integer $j > 0$,

$$\sum_{t=0}^{n-1} e^{it\omega_j} = \sum_{t=0}^{n-1} \exp\left\{ i2\pi j \frac{t}{n} \right\} = 0$$

Letting $\bar{X}$ denote the sample mean $n^{-1} \sum_{t=0}^{n-1} X_t$, we then have

$$nI(\omega_j) = \left| \sum_{t=0}^{n-1} (X_t - \bar{X}) e^{it\omega_j} \right|^2 = \sum_{t=0}^{n-1} (X_t - \bar{X}) e^{it\omega_j} \sum_{r=0}^{n-1} (X_r - \bar{X}) e^{-ir\omega_j}$$

By carefully working through the sums, one can transform this to

$$nI(\omega_j) = \sum_{t=0}^{n-1}(X_t - \bar{X})^2 + 2\sum_{k=1}^{n-1}\sum_{t=k}^{n-1}(X_t - \bar{X})(X_{t-k} - \bar{X})\cos(\omega_j k)$$

Now let

$$\hat{\gamma}(k) := \frac{1}{n}\sum_{t=k}^{n-1}(X_t - \bar{X})(X_{t-k} - \bar{X}), \qquad k = 0, 1, \ldots, n-1$$

This is the sample autocovariance function, the natural "plug-in estimator" of the *autocovariance function* γ

("Plug-in estimator" is an informal term for an estimator found by replacing expectations with sample means)

With this notation, we can now write

$$I(\omega_j) = \hat{\gamma}(0) + 2\sum_{k=1}^{n-1}\hat{\gamma}(k)\cos(\omega_j k)$$

Recalling our expression for f given *above*, we see that $I(\omega_j)$ is just a sample analog of $f(\omega_j)$

Calculation

Let's now consider how to compute the periodogram as defined in (4.120)

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

$$A_j := \sum_{t=0}^{n-1} a_t \exp\left\{i2\pi\frac{tj}{n}\right\}, \qquad j = 0, \ldots, n-1$$

With `numpy.fft.fft` imported as `fft` and $a_0, \ldots, a_{n-1}$ stored in NumPy array `a`, the function call `fft(a)` returns the values $A_0, \ldots, A_{n-1}$ as a NumPy array

It follows that, when the data $X_0, \ldots, X_{n-1}$ are stored in array `X`, the values $I(\omega_j)$ at the Fourier frequencies, which are given by

$$\frac{1}{n}\left|\sum_{t=0}^{n-1} X_t \exp\left\{i2\pi\frac{tj}{n}\right\}\right|^2, \qquad j = 0, \ldots, n-1$$

can be computed by `np.abs(fft(X))**2 / len(X)`

Note: The NumPy function `abs` acts elementwise, and correctly handles complex numbers (by computing their modulus, which is exactly what we need)

A function called `periodogram` that puts all this together can be found here

Let's generate some data for this function using the `ARMA` class from QuantEcon.py (see the *lecture on linear processes* for more details)

Here's a code snippet that, once the preceding code has been run, generates data from the process

$$X_t = 0.5X_{t-1} + \epsilon_t - 0.8\epsilon_{t-2} \tag{4.121}$$

where $\{\epsilon_t\}$ is white noise with unit variance, and compares the periodogram to the actual spectral density

```
import matplotlib.pyplot as plt
from quantecon import ARMA, periodogram

n = 40                          # Data size
phi, theta = 0.5, (0, -0.8)     # AR and MA parameters
lp = ARMA(phi, 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 it's not surprising that the estimate is poor

However, if we try again with `n = 1200` the outcome is not much better

The periodogram is far too irregular relative to the underlying spectral density

This brings us to our next topic

Smoothing

There are two related issues here

One is that, given the way the fast Fourier transform is implemented, the number of points ω at which $I(\omega)$ is estimated increases in line with the amount of data

In other words, although we have more data, we are also using it to estimate more values

A second issue is that densities of all types are fundamentally hard to estimate without parameteric 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

$$I_S(\omega_j) := \sum_{\ell=-p}^{p} w(\ell) I(\omega_{j+\ell}) \tag{4.122}$$

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 (4.122)

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 (4.121) — 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

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 (4.122)

The underlying assumption that makes this a good idea is that the true spectral density is relatively regular — the value of $I(\omega)$ is close to that of $I(\omega')$ when ω is close to ω'

This will not be true in all cases, but it is certainly true for white noise

For white noise, I is as regular as possible — *it is a constant function*

In this case, values of $I(\omega')$ at points ω' near to ω provided the maximum possible amount of information about the value $I(\omega)$

Another way to put this is that if I is relatively constant, then we can use a large amount of smoothing without introducing too much bias

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 partcular that $\{X_t\}$ is covariance stationary and AR(1), with

$$X_{t+1} = \mu + \phi X_t + \epsilon_{t+1} \tag{4.123}$$

where μ and $\phi \in (-1, 1)$ are unknown parameters and $\{\epsilon_t\}$ is white noise

It follows that if we regress X_{t+1} on X_t and an intercept, the residuals will approximate white noise

Let

- g be the spectral density of $\{\epsilon_t\}$ — a constant function, as discussed above
- I_0 be the periodogram estimated from the residuals — an estimate of g
- f be the spectral density of $\{X_t\}$ — the object we are trying to estimate

In view of *an earlier result* we obtained while discussing ARMA processes, f and g are related by

$$f(\omega) = \left| \frac{1}{1 - \phi e^{i\omega}} \right|^2 g(\omega) \tag{4.124}$$

This suggests that the recoloring step, which constructs an estimate I of f from I_0, should set

$$I(\omega) = \left| \frac{1}{1 - \hat{\phi} e^{i\omega}} \right|^2 I_0(\omega)$$

where $\hat{\phi}$ is the OLS estimate of ϕ

The code for `ar_periodogram()` — the third function in `estspec.py` — does exactly this. (See the code here)

The next figure shows realizations of the two kinds of smoothed periodograms

1. "standard smoothed periodogram", the ordinary smoothed periodogram, and
2. "AR smoothed periodogram", the pre-whitened and recolored one generated by `ar_periodogram()`

The periodograms are calculated from time series drawn from (4.123) with $\mu = 0$ and $\phi = -0.9$

Each time series is of length 150

The difference between the three subfigures is just randomness — each one uses a different draw of the time series

In all cases, periodograms are fit with the "hamming" window and window length of 65

Overall, the fit of the AR smoothed periodogram is much better, in the sense of being closer to the true spectral density

Exercises

Exercise 1

Replicate *this figure* (modulo randomness)

The model is as in equation (4.121) and there are 400 observations

For the smoothed periodogram, the windown type is "hamming"

Exercise 2

Replicate *this figure* (modulo randomness)

The model is as in equation (4.123), 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.

Solutions

```
import numpy as np
import matplotlib.pyplot as plt
from quantecon import ARMA, periodogram, ar_periodogram
```

Exercise 1

```
## Data
n = 400
phi = 0.5
theta = 0, -0.8
lp = ARMA(phi, theta)
X = lp.simulation(ts_length=n)

fig, ax = plt.subplots(3, 1, figsize=(10, 12))

for i, wl in enumerate((15, 55, 175)):  # window lengths

    x, y = periodogram(X)
    ax[i].plot(x, y, 'b-', lw=2, alpha=0.5, label='periodogram')

    x_sd, y_sd = lp.spectral_density(two_pi=False, res=120)
    ax[i].plot(x_sd, y_sd, 'r-', lw=2, alpha=0.8, label='spectral density')

    x, y_smoothed = periodogram(X, window='hamming', window_len=wl)
    ax[i].plot(x, y_smoothed, 'k-', lw=2, label='smoothed periodogram')
```

```
    ax[i].legend()
    ax[i].set_title('window length = {}'.format(wl))
plt.show()
```

Exercise 2

```
lp = ARMA(-0.9)
wl = 65


fig, ax = plt.subplots(3, 1, figsize=(10,12))

for i in range(3):
    X = lp.simulation(ts_length=150)
    ax[i].set_xlim(0, np.pi)

    x_sd, y_sd = lp.spectral_density(two_pi=False, res=180)
    ax[i].semilogy(x_sd, y_sd, 'r-', lw=2, alpha=0.75, label='spectral density
↪')

    x, y_smoothed = periodogram(X, window='hamming', window_len=wl)
    ax[i].semilogy(x, y_smoothed, 'k-', lw=2, alpha=0.75, label='standard␣
↪smoothed periodogram')

    x, y_ar = ar_periodogram(X, window='hamming', window_len=wl)
    ax[i].semilogy(x, y_ar, 'b-', lw=2, alpha=0.75, label='AR smoothed␣
↪periodogram')

    ax[i].legend(loc='upper left')
plt.show()
```

Additive Functionals

Contents

- *Additive Functionals*
  - *Overview*
  - *A Particular Additive Functional*
  - *Dynamics*
  - *Code*

Co-authors: Chase Coleman and Balint Szoke

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]*

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)

$$x_{t+1} = Ax_t + Bz_{t+1} \tag{4.125}$$

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 ν,
- the vector x_t, and
- the same Gaussian vector z_{t+1} that appears in the VAR (4.125)

In particular,

$$y_{t+1} - y_t = \nu + Dx_t + Fz_{t+1} \tag{4.126}$$

Here $y_0 \sim \mathcal{N}(\mu_{y0}, \Sigma_{y0})$ 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

$$\begin{bmatrix} 1 \\ x_{t+1} \\ y_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & A & 0 \\ \nu & D' & 1 \end{bmatrix} \begin{bmatrix} 1 \\ x_t \\ y_t \end{bmatrix} + \begin{bmatrix} 0 \\ B \\ F' \end{bmatrix} z_{t+1}$$

$$\begin{bmatrix} x_t \\ y_t \end{bmatrix} = \begin{bmatrix} 0 & I & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ x_t \\ y_t \end{bmatrix}$$

This can be written as

$$\begin{aligned} \hat{x}_{t+1} &= \hat{A}\hat{x}_t + \hat{B}z_{t+1} \\ \hat{y}_t &= \hat{D}\hat{x}_t \end{aligned}$$

which is a standard linear state space system

To study it, we could map it into an instance of LinearStateSpace from QuantEcon.py

We will in fact use a different set of code for simulation, for reasons described below

Dynamics

Let's run some simulations to build intuition In doing so we'll assume that z_{t+1} is scalar and that $\tilde{x}_t$ follows a 4th-order scalar autoregession

$$\tilde{x}_{t+1} = \phi_1 \tilde{x}_t + \phi_2 \tilde{x}_{t-1} + \phi_3 \tilde{x}_{t-2} + \phi_4 \tilde{x}_{t-3} + \sigma z_{t+1} \tag{4.127}$$

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 (4.127) is not a first order system like (4.125), 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 (4.125) – (4.126) 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 class AMF_LSS_VAR

The code is repeated here for convenience

```
"""
Code for simulating, visualizing and manipulating additive and
multiplicative functionals.

@authors: Chase Coleman, Balint Skoze, Tom Sargent

"""

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 norm, lognorm


class AMF_LSS_VAR:
    """
    This class transforms an additive (multipilcative)
    functional into a QuantEcon linear state space system.
    """

    def __init__(self, A, B, D, F=None, nu=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 nu
        if not np.any(nu):
            self.nu = np.zeros((self.nm, 1))
        elif type(nu) == float:
            self.nu = np.asarray([[nu]])
        elif len(nu.shape) == 1:
            self.nu = np.expand_dims(nu, 1)
        else:
            self.nu = nu

        if self.nu.shape[0] != self.D.shape[0]:
            raise ValueError("The dimension of nu is inconsistent with D!")

        # Construct BIG state space representation
        self.lss = self.construct_ss()

    def construct_ss(self):
        """
```

```
        This creates the state space representation that can be passed
        into the quantecon LSS class.
        """
        # Pull out useful info
        nx, nk, nm = self.nx, self.nk, self.nm
        A, B, D, F, nu = self.A, self.B, self.D, self.F, self.nu
        if self.add_decomp:
            nu, H, g = self.add_decomp
        else:
            nu, 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␣
↪x_{t+1}
        A4 = np.hstack([nu, ny0c, D, ny1m, ny0m])           # Transition for␣
↪y_{t+1}
        A5 = np.hstack([ny0c, ny0c, nyx0m, ny0m, ny1m])     # Transition for␣
↪m_{t+1}
        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])     #␣
↪Selector for x_{t}
        G2 = np.hstack([ny0c, ny0c, nyx0m, ny1m, ny0m])                #␣
↪Selector for y_{t}
        G3 = np.hstack([ny0c, ny0c, nyx0m, ny0m, ny1m])                #␣
↪Selector for martingale
        G4 = np.hstack([ny0c, ny0c, -g, ny0m, ny0m])                   #␣
↪Selector for stationary
        G5 = np.hstack([ny0c, nu, nyx0m, ny0m, ny0m])                  #␣
↪Selector for trend
        Gbar = np.vstack([G1, G2, G3, G4, G5])

        # Build H matrix for LSS
        Hbar = np.zeros((Gbar.shape[0], nk))
```

```
        # Build LSS type
        x0 = np.hstack([1, 0, nx0r, ny0r, ny0r])
        S0 = np.zeros((len(x0), len(x0)))
        lss = qe.lss.LinearStateSpace(Abar, Bbar, Gbar, Hbar, mu_0=x0, Sigma_
↪0=S0)

        return lss

    def additive_decomp(self):
        """
        Return values for the martingale decomposition
            - nu        : unconditional mean difference in Y
            - H         : coefficient for the (linear) martingale component␣
↪(kappa_a)
            - g         : coefficient for the stationary component g(x)
            - Y_0       : it should be the function of X_0 (for now set it to␣
↪0.0)
        """
        I = np.identity(self.nx)
        A_res = la.solve(I - self.A, I)
        g = self.D @ A_res
        H = self.F + self.D @ A_res @ self.B

        return self.nu, H, g

    def multiplicative_decomp(self):
        """
        Return values for the multiplicative decomposition (Example 5.4.4.)
            - nu_tilde  : eigenvalue
            - H         : vector for the Jensen term
        """
        nu, H, g = self.additive_decomp()
        nu_tilde = nu + (.5)*np.expand_dims(np.diag(H @ H.T), 1)

        return nu_tilde, H, g

    def loglikelihood_path(self, x, y):
        A, B, D, F = self.A, self.B, self.D, self.F
        k, T = y.shape
        FF = F @ F.T
        FFinv = la.inv(FF)
        temp = y[:, 1:] - y[:, :-1] - D @ x[:, :-1]
        obs =  temp * FFinv * temp
        obssum = np.cumsum(obs)
        scalar = (np.log(la.det(FF)) + k*np.log(2*np.pi))*np.arange(1, T)

        return -(.5)*(obssum + scalar)

    def loglikelihood(self, x, y):
        llh = self.loglikelihood_path(x, y)

        return llh[-1]
```

```
    def plot_additive(self, T, npaths=25, show_trend=True):
        """
        Plots for the additive decomposition

        """
        # Pull out right sizes so we know how to increment
        nx, nk, nm = self.nx, self.nk, self.nm

        # Allocate space (nm is the number of additive functionals - we want␣
↪npaths for each)
        mpath = np.empty((nm*npaths, T))
        mbounds = np.empty((nm*2, T))
        spath = np.empty((nm*npaths, T))
        sbounds = np.empty((nm*2, T))
        tpath = np.empty((nm*npaths, T))
        ypath = np.empty((nm*npaths, T))

        # Simulate for as long as we wanted
        moment_generator = self.lss.moment_sequence()
        # Pull out population moments
        for t in range (T):
            tmoms = next(moment_generator)
            ymeans = tmoms[1]
            yvar = tmoms[3]

            # Lower and upper bounds - for each additive functional
            for ii in range(nm):
                li, ui = ii*2, (ii+1)*2
                madd_dist = norm(ymeans[nx+nm+ii], np.sqrt(yvar[nx+nm+ii,␣
↪nx+nm+ii]))
                mbounds[li:ui, t] = madd_dist.ppf([0.01, .99])

                sadd_dist = norm(ymeans[nx+2*nm+ii], np.sqrt(yvar[nx+2*nm+ii,␣
↪nx+2*nm+ii]))
                sbounds[li:ui, t] = sadd_dist.ppf([0.01, .99])

        # Pull out paths
        for n in range(npaths):
            x, y = self.lss.simulate(T)
            for ii in range(nm):
                ypath[npaths*ii+n, :] = y[nx+ii, :]
                mpath[npaths*ii+n, :] = y[nx+nm + ii, :]
                spath[npaths*ii+n, :] = y[nx+2*nm + ii, :]
                tpath[npaths*ii+n, :] = y[nx+3*nm + ii, :]

        add_figs = []

        for ii in range(nm):
            li, ui = npaths*(ii), npaths*(ii+1)
            LI, UI = 2*(ii), 2*(ii+1)
            add_figs.append(self.plot_given_paths(T, ypath[li:ui,:],␣
↪mpath[li:ui,:], spath[li:ui,:],
```

```
                                                        tpath[li:ui,:],␣
↪mbounds[LI:UI,:], sbounds[LI:UI,:],
                                                        show_trend=show_trend))

            add_figs[ii].suptitle( r'Additive decomposition of $y_{%s}$' %␣
↪str(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
        nu_tilde, H, g = self.multiplicative_decomp()

        # Allocate space (nm is the number of functionals - we want npaths␣
↪for each)
        mpath_mult = np.empty((nm*npaths, T))
        mbounds_mult = np.empty((nm*2, T))
        spath_mult = np.empty((nm*npaths, T))
        sbounds_mult = np.empty((nm*2, T))
        tpath_mult = np.empty((nm*npaths, T))
        ypath_mult = np.empty((nm*npaths, T))

        # Simulate for as long as we wanted
        moment_generator = self.lss.moment_sequence()
        # Pull out population moments
        for t in range (T):
            tmoms = next(moment_generator)
            ymeans = tmoms[1]
            yvar = tmoms[3]

            # Lower and upper bounds - for each multiplicative functional
            for ii in range(nm):
                li, ui = ii*2, (ii+1)*2
                Mdist = lognorm(np.asscalar(np.sqrt(yvar[nx+nm+ii,␣
↪nx+nm+ii])),
                                scale=np.asscalar( np.exp( ymeans[nx+nm+ii]- \
                                                t*(.5)*np.expand_dims(np.
↪diag(H @ H.T),1)[ii])))
                Sdist = lognorm(np.asscalar(np.sqrt(yvar[nx+2*nm+ii,␣
↪nx+2*nm+ii])),
                                scale = np.asscalar( np.exp(-
↪ymeans[nx+2*nm+ii])))
                mbounds_mult[li:ui, t] = Mdist.ppf([.01, .99])
                sbounds_mult[li:ui, t] = Sdist.ppf([.01, .99])

        # Pull out paths
```

```
        for n in range(npaths):
            x, y = self.lss.simulate(T)
            for ii in range(nm):
                ypath_mult[npaths*ii+n, :] = np.exp(y[nx+ii, :])
                mpath_mult[npaths*ii+n, :] = np.exp(y[nx+nm + ii, :] - np.
↪arange(T)*(.5)*np.expand_dims(np.diag(H @ H.T),1)[ii])
                spath_mult[npaths*ii+n, :] = 1/np.exp(-y[nx+2*nm + ii, :])
                tpath_mult[npaths*ii+n, :] = np.exp(y[nx+3*nm + ii, :] + np.
↪arange(T)*(.5)*np.expand_dims(np.diag(H @ H.T),1)[ii])

        mult_figs = []

        for ii in range(nm):
            li, ui = npaths*(ii), npaths*(ii+1)
            LI, UI = 2*(ii), 2*(ii+1)

            mult_figs.append(self.plot_given_paths(T, ypath_mult[li:ui,:],␣
↪mpath_mult[li:ui,:],
                                                   spath_mult[li:ui,:], tpath_
↪mult[li:ui,:],
                                                   mbounds_mult[LI:UI,:],␣
↪sbounds_mult[LI:UI,:], 1,
                                                   show_trend=show_trend))
            mult_figs[ii].suptitle( r'Multiplicative decomposition of $y_{%s}$
↪' % str(ii+1), fontsize=14)

        return mult_figs

    def plot_martingales(self, T, npaths=25):

        # Pull out right sizes so we know how to increment
        nx, nk, nm = self.nx, self.nk, self.nm
        # Matrices for the multiplicative decomposition
        nu_tilde, H, g = self.multiplicative_decomp()

        # Allocate space (nm is the number of functionals - we want npaths␣
↪for each)
        mpath_mult = np.empty((nm*npaths, T))
        mbounds_mult = np.empty((nm*2, T))

        # Simulate for as long as we wanted
        moment_generator = self.lss.moment_sequence()
        # Pull out population moments
        for t in range (T):
            tmoms = next(moment_generator)
            ymeans = tmoms[1]
            yvar = tmoms[3]

            # Lower and upper bounds - for each functional
            for ii in range(nm):
                li, ui = ii*2, (ii+1)*2
                Mdist = lognorm(np.asscalar(np.sqrt(yvar[nx+nm+ii,␣
↪nx+nm+ii])),
```

```
                                scale=np.asscalar( np.exp( ymeans[nx+nm+ii]- \
                                                t*(.5)*np.expand_dims(np.
↪diag(H @ H.T),1)[ii])))
                mbounds_mult[li:ui, t] = Mdist.ppf([.01, .99])

        # Pull out paths
        for n in range(npaths):
            x, y = self.lss.simulate(T)
            for ii in range(nm):
                mpath_mult[npaths*ii+n, :] = np.exp(y[nx+nm + ii, :] - np.
↪arange(T)*(.5)*np.expand_dims(np.diag(H @ H.T),1)[ii])

        mart_figs = []

        for ii in range(nm):
            li, ui = npaths*(ii), npaths*(ii+1)
            LI, UI = 2*(ii), 2*(ii+1)
            mart_figs.append(self.plot_martingale_paths(T, mpath_mult[li:ui,
↪:],
                                                        mbounds_mult[LI:UI,:],
↪ horline=1))
            mart_figs[ii].suptitle(r'Martingale components for many paths of
↪$y_{%s}$' % str(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=r"$y_t$", color="k")
        ax[0, 0].plot(trange, mpath[0, :], label=r"$m_t$", color="m")
        ax[0, 0].plot(trange, spath[0, :], label=r"$s_t$", color="g")
        if show_trend:
            ax[0, 0].plot(trange, tpath[0, :], label=r"$t_t$", color="r")
        ax[0, 0].axhline(horline, color="k", linestyle = "-.")
        ax[0, 0].set_title("One Path of All Variables")
        ax[0, 0].legend(loc="upper left")

        # Plot Martingale Component
        ax[0, 1].plot(trange, mpath[0, :], "m")
        ax[0, 1].plot(trange, mpath.T, alpha=0.45, color="m")
        ub = mbounds[1, :]
        lb = mbounds[0, :]
        ax[0, 1].fill_between(trange, lb, ub, alpha=0.25, color="m")
```

```
        ax[0, 1].set_title("Martingale Components for Many Paths")
        ax[0, 1].axhline(horline, color="k", linestyle = "-.")

        # Plot Stationary Component
        ax[1, 0].plot(spath[0, :], color="g")
        ax[1, 0].plot(spath.T, alpha=0.25, color="g")
        ub = sbounds[1, :]
        lb = sbounds[0, :]
        ax[1, 0].fill_between(trange, lb, ub, alpha=0.25, color="g")
        ax[1, 0].axhline(horline, color="k", linestyle = "-.")
        ax[1, 0].set_title("Stationary Components for Many Paths")

        # Plot Trend Component
        if show_trend:
            ax[1, 1].plot(tpath.T, color="r")
        ax[1, 1].set_title("Trend Components for Many Paths")
        ax[1, 1].axhline(horline, color="k", linestyle = "-.")

        return fig

    def plot_martingale_paths(self, T, mpath, mbounds, horline=1, show_trend␣
↪= False):
        # Allocate space
        trange = np.arange(T)

        # Create figure
        fig, ax = plt.subplots(1,1, figsize=(10,6))

        # Plot Martingale Component
        ub = mbounds[1, :]
        lb = mbounds[0, :]
        ax.fill_between(trange, lb, ub, color="#ffccff")
        ax.axhline(horline, color="k", linestyle = "-.")
        ax.plot(trange, mpath.T, linewidth=0.25, color="#4c4c4c")

        return fig
```

For now, we just plot y_t and x_t, postponing until later a desciption of exactly how we compute them

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

phi_1, phi_2, phi_3, phi_4 = 0.5, -0.2, 0, 0.5
sigma = 0.01
nu = 0.01 # Growth rate

# A matrix should be n x n
A = np.array([[phi_1, phi_2, phi_3, phi_4],
              [1,     0,     0,     0],
              [0,     1,     0,     0],
              [0,     0,     1,     0]])

# B matrix should be n x k
```

```
B = np.array([[sigma, 0, 0, 0]]).T

D = np.array([1, 0, 0, 0]).dot(A)
F = np.array([1, 0, 0, 0]).dot(B)

amf = AMF_LSS_VAR(A, B, D, F, nu=nu)

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 *[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 (4.125) and (4.126),

we first construct the matrices

$$H := F + B'(I - A')^{-1}D$$
$$g := D'(I - A)^{-1}$$

Then the Hansen-Scheinkman *[HS09]* decomposition is

$$y_t = \underbrace{t\nu}_{\text{trend component}} + \overbrace{\sum_{j=1}^{t} Hz_j}^{\text{Martingale component}} - \underbrace{gx_t}_{\text{stationary component}} + \overbrace{gx_0 + y_0}^{\text{initial conditions}}$$

At this stage you should pause and verify that $y_{t+1} - y_t$ satisfies (4.126)

It is convenient for us to introduce the following notation:

- $\tau_t = \nu t$, a linear, deterministic trend
- $m_t = \sum_{j=1}^{t} Hz_j$, a martingale with time $t+1$ increment Hz_{t+1}
- $s_t = gx_t$, an (asymptotically) stationary component

We want to characterize and simulate components τ_t, m_t, s_t of the decomposition.

A convenient way to do this is to construct an appropriate instance of a *linear state space system* by using LinearStateSpace from QuantEcon.py

This will allow us to use the routines in LinearStateSpace to study dynamics

To start, observe that, under the dynamics in (4.125) and (4.126) and with the definitions just given,

$$\begin{bmatrix} 1 \\ t+1 \\ x_{t+1} \\ y_{t+1} \\ m_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & A & 0 & 0 \\ \nu & 0 & D' & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ x_t \\ y_t \\ m_t \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ B \\ F' \\ H' \end{bmatrix} z_{t+1}$$

and

$$\begin{bmatrix} x_t \\ y_t \\ \tau_t \\ m_t \\ s_t \end{bmatrix} = \begin{bmatrix} 0 & 0 & I & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & \nu & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -g & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ x_t \\ y_t \\ m_t \end{bmatrix}$$

With

$$\tilde{x} := \begin{bmatrix} 1 \\ t \\ x_t \\ y_t \\ m_t \end{bmatrix} \quad \text{and} \quad \tilde{y} := \begin{bmatrix} x_t \\ y_t \\ \tau_t \\ m_t \\ s_t \end{bmatrix}$$

we can write this as the linear state space system

$$\tilde{x}_{t+1} = \tilde{A}\tilde{x}_t + \tilde{B}z_{t+1}$$
$$\tilde{y}_t = \tilde{D}\tilde{x}_t$$

By picking out components of $\tilde{y}_t$, we can track all variables of interest

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

```
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 LSS 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

$$\frac{M_t}{M_0} = \exp(t\nu) \exp\Big(\sum_{j=1}^{t} H \cdot Z_j\Big) \exp\Big(D'(I-A)^{-1}x_0 - D'(I-A)^{-1}x_t\Big)$$

or

$$\frac{M_t}{M_0} = \exp\left(\tilde{\nu} t\right) \left(\frac{\widetilde{M}_t}{\widetilde{M}_0}\right) \left(\frac{\tilde{e}(X_0)}{\tilde{e}(x_t)}\right)$$

where

$$\tilde{\nu} = \nu + \frac{H \cdot H}{2}, \quad \widetilde{M}_t = \exp\Big(\sum_{j=1}^{t}\Big(H \cdot z_j - \frac{H \cdot H}{2}\Big)\Big), \quad \widetilde{M}_0 = 1$$

and

$$\tilde{e}(x) = \exp[g(x)] = \exp\big[D'(I-A)^{-1}x\big]$$

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

```
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 LSS 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 $\widetilde{M}_t$ of the multplicative decomposition has a peculiar property

- While $E_0 \widetilde{M}_t = 1$ for all $t \geq 0$, nevertheless . . .
- As $t \to +\infty$, $\widetilde{M}_t$ converges to zero almost surely

The following simulation of many paths of $\widetilde{M}_t$ illustrates this property

```
np.random.seed(10021987)
amf.plot_martingales(12000)
plt.show()
```

Here's the resulting figure:

Multiplicative Functionals

Contents

- *Multiplicative Functionals*
  - *Overview*
  - *A Log-Likelihood Process*
  - *Benefits from Reduced Aggregate Fluctuations*

Co-authors: Chase Coleman and Balint Szoke

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

- A version of Robert E. Lucas's *[Jr03]* and Thomas Tallarini's *[Tal00]* approaches to measuring the benefits of moderating aggregate fluctuations

A Log-Likelihood Process

Consider a vector of additive functionals $\{y_t\}_{t=0}^{\infty}$ described by

$$\begin{aligned} x_{t+1} &= Ax_t + Bz_{t+1} \\ y_{t+1} - y_t &= Dx_t + Fz_{t+1}, \end{aligned}$$

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,

$$x_{t+1} = \left(A - BF^{-1}D\right) x_t + BF^{-1}\left(y_{t+1} - y_t\right),$$

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 θ 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

$$\begin{aligned} \log L_t(\theta) = &-\frac{1}{2}\sum_{j=1}^{t}(y_j - y_{j-1} - Dx_{j-1})'(FF')^{-1}(y_j - y_{j-1} - Dx_{j-1}) \\ &- \frac{t}{2}\log\det(FF') - \frac{kt}{2}\log(2\pi) \end{aligned}$$

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 θ_o denote the "true" values of θ, 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'll do this by formulating the additive functional as a linear state space model and putting the LinearStateSpace class to work

```
"""
Code for simulating, visualizing and manipulating *scalar* additive␣
↪and
multiplicative functionals.

@authors: Chase Coleman, Balint Skoze, Tom Sargent

"""


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, nu, 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.)
            - nu        : unconditional mean difference in Y
            - H         : coefficient for the (linear) martingale␣
↪component (kappa_a)
            - g         : coefficient for the stationary component␣
↪g(x)
            - Y_0       : it should be the function of X_0 (for now␣
↪set it to 0.0)
        """
        A_res = 1 / (1 - self.A)
        g = self.D * A_res
        H = self.F + self.D * A_res * self.B

        return self.nu, H, g

    def multiplicative_decomp(self):
```

```
        """
        Return values for the multiplicative decomposition (Example␣
↪5.4.4.)
            - nu_tilde  : eigenvalue
            - H         : vector for the Jensen term
        """
        nu, H, g = self.additive_decomp()
        nu_tilde = nu + (.5)*H**2

        return nu_tilde, H, g

    def loglikelihood_path(self, x, y):
        A, B, D, F = self.A, self.B, self.D, self.F
        T = y.T.size
        FF = F**2
        FFinv = 1/FF
        temp = y[1:] - y[:-1] - D*x[:-1]
        obs =  temp * FFinv * temp
        obssum = np.cumsum(obs)
        scalar = (np.log(FF) + np.log(2*np.pi))*np.arange(1,T)

        return (-0.5)*(obssum + scalar)

    def loglikelihood(self, x, y):
        llh = self.loglikelihood_path(x, y)

        return llh[-1]
```

The heavy lifting is done inside the *AMF_LSS_VAR* class

The following code adds some simple functions that make it straightforward to generate sample paths from an instance of *AMF_LSS_VAR*

```
import matplotlib.pyplot as plt
import numpy as np
import scipy as sp
import scipy.linalg as la
import seaborn as sb
import quantecon as qe
from scipy.stats import lognorm, norm

def simulate_xy(amf, T):
    "Simulate individual paths."
    foo, bar = amf.lss.simulate(T)
    x = bar[0, :]
    y = bar[1, :]

    return x, y

def simulate_paths(amf, T=150, I=5000):
    "Simulate multiple independent paths."

    # Allocate space
```

```
    storeX = np.empty((I, T))
    storeY = np.empty((I, T))

    for i in range(I):
        # Do specific simulation
        x, y = simulate_xy(amf, T)

        # Fill in our storage matrices
        storeX[i, :] = x
        storeY[i, :] = y

    return storeX, storeY

def population_means(amf, T=150):
    # Allocate Space
    xmean = np.empty(T)
    ymean = np.empty(T)

    # Pull out moment generator
    moment_generator = amf.lss.moment_sequence()

    for tt in range (T):
        tmoms = next(moment_generator)
        ymeans = tmoms[1]
        xmean[tt] = ymeans[0]
        ymean[tt] = ymeans[1]

    return xmean, ymean
```

Now that we have these functions in our took kit, let's apply them to run some simulations

In particular, let's use our program to generate $I = 5000$ sample paths of length $T = 150$, labeled $\{x_t^i, y_t^i\}_{t=0}^{\infty}$ 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=r'$\mathbb{E} x_t$', color="k")
ax[0].set_title(r'$x_t$')
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=r'$\mathbb{E} y_t$', color="k")
ax[1].set_title(r'$y_t$')
ax[1].set_xlim((0, T))
ax[1].legend(loc=0)

plt.show()
```

Here's the resulting figure

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 $\{x_t^i, y_t^i\}_{t=0}^T$ that we have already computed

We now want to simulate $I = 5000$ paths of $\{\log L_t^i \mid \theta_o\}_{t=1}^T$

- For each path, we compute $\log L_T^i / T$
- We also compute $\frac{1}{I}\sum_{i=1}^I \log L_T^i / T$

Then we to compare these objects

Below we plot the histogram of $\log L_T^i / T$ for realizations $i = 1, \ldots, 5000$

```
def simulate_likelihood(amf, Xit, Yit):
    # Get size
    I, T = Xit.shape

    # Allocate space
    LLit = np.empty((I, T-1))

    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)
fig.suptitle(r"Distribution of $\frac{1}{T} \log L_{T}  \mid \theta_0$",␣
↪fontsize=14)

plt.show()
```

Here's the resulting figure

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 Δy_{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

```
normdist = sp.stats.norm(0,F)
mult = 1.175
print('The pdf at +/- {} sigma takes the value:  {}'.format(mult, normdist.
↪pdf(mult*F)))
print('Probability of dL being larger than 1 is approx: {}'.format( 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('Fraction of dlogL being nonnegative in the sample is: {}'.format(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 Δy_{t+1}

```
xgrid = np.linspace(-1,1,100)
plt.plot(xgrid, normdist.pdf(xgrid))
plt.title(r'Conditional pdf $f(\Delta y_{t+1}  \mid x_t)$')
print('The pdf at +/- one sigma takes the value: {}'.format(normdist.pdf(F)) )
plt.show()
```

Here's the resulting figure

```
The pdf at +/- one sigma takes the value: 1.2098536225957168
```

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 we computed above

This is because we want to generate data under the *:math:'theta_o' probability model* but evaluate the likelihood under the θ_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

Here's the code

```
# 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_o
amf2 = AMF_LSS_VAR(A2, B2, D2, F=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)

fig.suptitle(r"Distribution of $\frac{1}{T} \log L_{T}  \mid \theta_1$",␣
↪fontsize=14)
plt.show()
```

The resulting figure looks like this

Let's see a histogram of the log-likelihoods under the true and the alternative model (same sample paths)

```
fig, ax = plt.subplots(figsize=(8, 6))

plt.hist(LLT, bins=50, alpha=0.5, label='True', normed=True)
plt.hist(LLT2, bins=50, alpha =0.5, label='Alternative', normed=True)
plt.vlines(np.mean(LLT), 0, 10, color='k', linestyle="--", linewidth= 4)
plt.vlines(np.mean(LLT2), 0, 10, color='k', linestyle="--", linewidth= 4)
plt.legend(loc='best')

plt.show()
```

Now we'll plot the histogram of the difference in log likelihood ratio

```
LLT_diff = LLT - LLT2

fig, ax = plt.subplots(figsize=(8, 6))

ax.hist(LLT_diff, bins=50)
fig.suptitle(r"$\frac{1}{T}\left[\log (L_T^i  \mid \theta_0) - \log (L_T^i ␣
↪\mid \theta_1)\right]$", fontsize=15)

plt.show()
```

The resulting figure is as follows

Interpretation

These histograms of log likelihood ratios illustrate important features of **likelihood ratio tests** as tools for discriminating between statistical models

- The log likeklihood 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, we'll use some of the code prepared in this lecture to illustrate mistake probabilities)

Benefits from Reduced Aggregate Fluctuations

Now let's 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]*, *[Jr03]*)

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

$$\log V_t - \log c_t = U \cdot x_t + \mathsf{u} \tag{4.128}$$

where

$$U = \exp(-\delta)\left[I - \exp(-\delta)A'\right]^{-1} D$$

and

$$\mathsf{u} = \frac{\exp(-\delta)}{1-\exp(-\delta)}\nu + \frac{(1-\gamma)}{2}\frac{\exp(-\delta)}{1-\exp(-\delta)}\left|D'\left[I - \exp(-\delta)A\right]^{-1} B + F\right|^2,$$

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

$$\frac{c_t}{c_0} = \exp(\tilde{\nu} t)\left(\frac{\tilde{M}_t}{\tilde{M}_0}\right)\left(\frac{\tilde{e}(x_0)}{\tilde{e}(x_t)}\right) \tag{4.129}$$

where $\left(\frac{\tilde{M}_t}{\tilde{M}_0}\right)$ is a likelihood ratio process and $\tilde{M}_0 = 1$

At this point, as an exercise, we ask the reader please to verify the follow formulas for $\tilde{\nu}$ and $\tilde{e}(x_t)$ as functions of A, B, D, F:

$$\tilde{\nu} = \nu + \frac{H \cdot H}{2}$$

and

$$\tilde{e}(x) = \exp[g(x)] = \exp\left[D'(I-A)^{-1}x\right]$$

Simulating a likelihood ratio process again

Next, we want a program to simulate the likelihood ratio process $\{\tilde{M}_t\}_{t=0}^{\infty}$

In particular, we want to simulate 5000 sample paths of length $T = 1000$ 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 $\tilde{M}_T^i$ for $T = 1000$

Here is code that accomplishes these tasks

```
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,.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("\t ({}, {}, {})".format(np.min(amcT), np.mean(amcT), np.max(amcT)))

print("The (min, mean, max) of multiplicative Martingale component in period␣
↪T is")
print("\t ({}, {}, {})".format(np.min(mmcT), np.mean(mmcT), np.max(mmcT)))
```

Here's 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

Here's a histogram of the additive martingale component

```
fig, ax = plt.subplots(figsize=(8, 6))

ax.hist(amcT, bins=25, normed=True)
fig.suptitle("Histogram of Additive Martingale Component", fontsize=14)

plt.show()
```

Here's a histogram of the multiplicative martingale component

```
fig, ax = plt.subplots(figsize=(8, 6))

ax.hist(mmcT, bins=25, normed=True)
fig.suptitle("Histogram of Multiplicative Martingale Component", fontsize=14)

plt.show()
```

Representing the likelihood ratio process

The likelihood ratio process $\{\widetilde{M}_t\}_{t=0}^{\infty}$ can be represented as

$$\widetilde{M}_t = \exp\left(\sum_{j=1}^{t}\left(H \cdot z_j - \frac{H \cdot H}{2}\right)\right), \quad \widetilde{M}_0 = 1,$$

where $H = [F + B'(I - A')^{-1}D]$

It follows that $\log \widetilde{M}_t \sim \mathcal{N}(-\frac{tH \cdot H}{2}, tH \cdot H)$ and that consequently $\widetilde{M}_t$ is log normal

Let's plot the probability density functions for $\log \widetilde{M}_t$ for $t = 100, 500, 1000, 10000, 100000$

Then let's use the plots to investigate how these densities evolve through time

We will plot the densities of $\log \widetilde{M}_t$ for different values of t

Note: `scipy.stats.lognorm` expects you to pass the standard deviation first $(tH \cdot H)$ and then the exponent of the mean as a keyword argument `scale` (`scale=`$\exp(-tH \cdot H/2)$)

This is peculiar, so make sure you are careful in working with the log normal distribution

Here is some code that tackles these tasks

```
def Mtilde_t_density(amf, t, xmin=1e-8, xmax=5.0, npts=5000):

    # Pull out the multiplicative decomposition
    nutilde, 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
    nutilde, 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("Density for time {}".format(times_to_plot[it]))
```

```
    ax[it].fill_between(x, np.zeros_like(pdf), pdf)

fig.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

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]*, *[Jr03]*) 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 (4.128) 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 (4.129)

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, ν, but also for the case that $A, B, D, F = [0, 0, 0, 0]$ and $\nu = \tilde{\nu}$

Here's our code

```
def Uu(amf, delta, gamma):
    A, B, D, F, nu = amf.A, amf.B, amf.D, amf.F, amf.nu
    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)*(vect**2))

    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 paramaters
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

$$\log V_0^r = \log c_0^r + U^r x_0 + u^r$$
$$\log V_0^d = \log c_0^d + U^d x_0 + u^d$$

We look for the ratio $\frac{c_0^r - c_0^d}{c_0^r}$ that makes $\log V_0^r - \log V_0^d = 0$

$$\underbrace{\log V_0^r - \log V_0^d}_{=0} + \log c_0^d - \log c_0^r = (U^r - U^d)x_0 + u^r - u^d$$

$$\frac{c_0^d}{c_0^r} = \exp\left((U^r - U^d)x_0 + u^r - u^d\right)$$

Hence, the implied percentage reduction in c_0 that the representative consumer would accept is given by

$$\frac{c_0^r - c_0^d}{c_0^r} = 1 - \exp\left((U^r - U^d)x_0 + u^r - u^d\right)$$

Let's compute this

```
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) )
```

If we print this value out 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*

Classical Control with Linear Algebra

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 we'll put to work here is LU decomposition

We'll begin with discrete horizon problems

Then we'll view infinite horizon problems as appropriate limits of these finite horizon problems

Later, we will examine the close connection between LQ control and least squares prediction and filtering problems

These classes of problems are connected in the sense that to solve each, essentially the same mathematics is used

References

Useful references include *[Whi63]*, *[HS80]*, *[Orf88]*, *[AP91]*, and *[Mut60]*

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

$$\max_{\{y_t\}} \lim_{N\to\infty} \sum_{t=0}^{N} \beta^t \left\{ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} \left[d(L) y_t\right]^2 \right\}, \tag{4.130}$$

where

- h is a positive parameter and $\beta \in (0,1)$ is a discount factor

- $\{a_t\}_{t\geq 0}$ is a sequence of exponential order less than $\beta^{-1/2}$, by which we mean $\lim_{t\to\infty} \beta^{\frac{t}{2}} a_t = 0$

Maximization in (4.130) 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

$$\pi_t = p_t q_t - c q_t - \gamma(q_{t+1} - q_t)^2 \quad \text{with} \quad p_t = \alpha_0 - \alpha_1 q_t + d_t$$

In this expression, q_t is output, c is average cost of production, and d_t is a demand shock

The term $\gamma(q_{t+1} - q_t)^2$ represents adjustment costs

You will be able to confirm that the objective function can be rewritten as (4.130) 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]*

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 (4.130) are not sufficient for maximizing (4.130))

We begin by

1. fixing $N > m$,
2. differentiating the finite version of (4.130) with respect to $y_0, y_1, \ldots, y_N$, and
3. setting these derivatives to zero

For $t = 0, \ldots, N - m$ these first-order necessary conditions are the *Euler equations*

For $t = N - m + 1, \ldots, N$, the first-order conditions are a set of *terminal conditions*

Consider the term

$$
\begin{aligned}
J &= \sum_{t=0}^{N} \beta^t [d(L) y_t][d(L) y_t] \\
&= \sum_{t=0}^{N} \beta^t \left(d_0\, y_t + d_1\, y_{t-1} + \cdots + d_m\, y_{t-m}\right)\left(d_0\, y_t + d_1\, y_{t-1} + \cdots + d_m\, y_{t-m}\right)
\end{aligned}
$$

Differentiating J with respect to y_t for $t = 0,\ 1,\ \ldots,\ N - m$ gives

$$
\begin{aligned}
\frac{\partial J}{\partial y_t} &= 2\beta^t\, d_0\, d(L) y_t + 2\beta^{t+1}\, d_1\, d(L) y_{t+1} + \cdots + 2\beta^{t+m}\, d_m\, d(L) y_{t+m} \\
&= 2\beta^t \left(d_0 + d_1\, \beta L^{-1} + d_2\, \beta^2\, L^{-2} + \cdots + d_m\, \beta^m\, L^{-m}\right) d(L) y_t
\end{aligned}
$$

We can write this more succinctly as

$$
\frac{\partial J}{\partial y_t} = 2\beta^t\, d(\beta L^{-1})\, d(L) y_t \tag{4.131}
$$

Differentiating J with respect to y_t for $t = N - m + 1, \ldots, N$ gives

$$
\begin{aligned}
\frac{\partial J}{\partial y_N} &= 2\beta^N\, d_0\, d(L) y_N \\
\frac{\partial J}{\partial y_{N-1}} &= 2\beta^{N-1} \left[d_0 + \beta\, d_1\, L^{-1}\right] d(L) y_{N-1} \\
\vdots \quad & \quad \vdots \\
\frac{\partial J}{\partial y_{N-m+1}} &= 2\beta^{N-m+1} \left[d_0 + \beta L^{-1}\, d_1 + \cdots + \beta^{m-1}\, L^{-m+1}\, d_{m-1}\right] d(L) y_{N-m+1}
\end{aligned} \tag{4.132}
$$

With these preliminaries under our belts, we are ready to differentiate (4.130)

Differentiating (4.130) with respect to y_t for $t = 0, \ldots, N - m$ gives the Euler equations

$$
\left[h + d\,(\beta L^{-1})\, d(L)\right] y_t = a_t, \quad t = 0,\ 1,\ \ldots, N - m \tag{4.133}
$$

The system of equations (4.133) form a $2 \times m$ order linear *difference equation* that must hold for the values of t indicated.

Differentiating (4.130) with respect to y_t for $t = N - m + 1, \ldots, N$ gives the terminal conditions

$$
\begin{aligned}
\beta^N (a_N - h y_N - d_0\, d(L) y_N) &= 0 \\
\beta^{N-1} \left(a_{N-1} - h y_{N-1} - \left(d_0 + \beta\, d_1\, L^{-1}\right) d(L)\, y_{N-1}\right) &= 0 \\
&\vdots \\
\beta^{N-m+1} \left(a_{N-m+1} - h y_{N-m+1} - (d_0 + \beta L^{-1} d_1 + \cdots + \beta^{m-1} L^{-m+1} d_{m-1}) d(L) y_{N-m+1}\right) &= 0
\end{aligned} \tag{4.134}
$$

In the finite N problem, we want simultaneously to solve (4.133) subject to the m initial conditions $y_{-1}, \ldots, y_{-m}$ and the m terminal conditions (4.134)

These conditions uniquely pin down the solution of the finite N problem

That is, for the finite N problem, conditions (4.133) and (4.134) 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

$$\begin{aligned} \left[h + d\left(\beta L^{-1}\right) d\left(L\right)\right] y_t &= a_t, \quad t = 0,\ 1,\ \ldots,\ N-1 \\ \beta^N \left[a_N - h\, y_N - d_0\, d\left(L\right) y_N\right] &= 0 \end{aligned} \tag{4.135}$$

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 (4.135) as the matrix equation

$$\begin{bmatrix} (\phi_0 - d_1^2) & \phi_1 & 0 & 0 & \ldots & \ldots & 0 \\ \beta\phi_1 & \phi_0 & \phi_1 & 0 & \ldots & \ldots & 0 \\ 0 & \beta\phi_1 & \phi_0 & \phi_1 & \ldots & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \ldots & \ldots & \ldots & \beta\phi_1 & \phi_0 & \phi_1 \\ 0 & \ldots & \ldots & \ldots & 0 & \beta\phi_1 & \phi_0 \end{bmatrix} \begin{bmatrix} y_N \\ y_{N-1} \\ y_{N-2} \\ \vdots \\ y_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} a_N \\ a_{N-1} \\ a_{N-2} \\ \vdots \\ a_1 \\ a_0 - \phi_1 y_{-1} \end{bmatrix} \tag{4.136}$$

or

$$W \bar{y} = \bar{a} \tag{4.137}$$

Notice how we have chosen to arrange the y_t's in reverse time order.

The matrix W on the left side of (4.136) 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 β time the superdiagonal elements

The solution of (4.137) can be expressed in the form

$$\bar{y} = W^{-1}\bar{a} \tag{4.138}$$

which represents each element y_t of $\bar{y}$ as a function of the entire vector $\bar{a}$

That is, y_t is a function of past, present, and future values of a‘s, as well as of the initial condition y_{-1}

An Alternative Representation

An alternative way to express the solution to (4.136) or (4.137) 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 trangular matrix
- U is an $(N+1) \times (N+1)$ upper trangular matrix.

The factorization can be normalized so that the diagonal elements of U are unity

Using the LU representation in (4.138), we obtain

$$U\bar{y} = L^{-1}\bar{a} \tag{4.139}$$

Since L^{-1} is lower trangular, this representation expresses y_t as a function of

- lagged y‘s (via the term $U\bar{y}$), and
- current and future a’s (via the term $L^{-1}\bar{a}$)

Because there are zeros everywhere in the matrix on the left of (4.136) except on the diagonal, superdiagonal, and subdiagonal, the LU decomposition takes

- L to be zero except in the diagional and the leading subdiagonal
- U to be zero except on the diagonal and the superdiagional

Thus, (4.139) has the form

$$\begin{bmatrix} 1 & U_{12} & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & U_{23} & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & U_{34} & \dots & 0 & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 & U_{N,N+1} \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} y_N \\ y_{N-1} \\ y_{N-2} \\ y_{N-3} \\ \vdots \\ y_1 \\ y_0 \end{bmatrix} =$$

$$\begin{bmatrix} L^{-1}_{11} & 0 & 0 & \dots & 0 \\ L^{-1}_{21} & L^{-1}_{22} & 0 & \dots & 0 \\ L^{-1}_{31} & L^{-1}_{32} & L^{-1}_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L^{-1}_{N,1} & L^{-1}_{N,2} & L^{-1}_{N,3} & \dots & 0 \\ L^{-1}_{N+1,1} & L^{-1}_{N+1,2} & L^{-1}_{N+1,3} & \dots & L^{-1}_{N+1\,N+1} \end{bmatrix} \begin{bmatrix} a_N \\ a_{N-1} \\ a_{N-2} \\ \vdots \\ a_1 \\ a_0 - \phi_1 y_{-1} \end{bmatrix}$$

where L^{-1}_{ij} is the (i,j) element of L^{-1} and U_{ij} is the (i,j) element of U

Note how the left side for a given t involves y_t and one lagged value y_{t-1} while the right side involves all future values of the forcing process $a_t, a_{t+1}, \dots, a_N$

Additional Lag Terms

We briefly indicate how this approach extends to the problem with $m > 1$

Assume that $\beta = 1$ and let D_{m+1} be the $(m+1)\times(m+1)$ symmetric matrix whose elements are determined from the following formula:

$$D_{jk} = d_0 d_{k-j} + d_1 d_{k-j+1} + \dots + d_{j-1} d_{k-1}, \qquad k \geq j$$

Let I_{m+1} be the $(m+1)\times(m+1)$ identity matrix

Let ϕ_j be the coefficients in the expansion $\phi(L) = h + d(L^{-1})d(L)$

Then the first order conditions (4.133) and (4.134) can be expressed as:

$$(D_{m+1} + hI_{m+1}) \begin{bmatrix} y_N \\ y_{N-1} \\ \vdots \\ y_{N-m} \end{bmatrix} = \begin{bmatrix} a_N \\ a_{N-1} \\ \vdots \\ a_{N-m} \end{bmatrix} + M \begin{bmatrix} y_{N-m+1} \\ y_{N-m-2} \\ \vdots \\ y_{N-2m} \end{bmatrix}$$

where M is $(m+1)\times m$ and

$$M_{ij} = \begin{cases} D_{i-j,\,m+1} \text{ for } i > j \\ 0 \text{ for } i \leq j \end{cases}$$

$$\begin{aligned} \phi_m y_{N-1} + \phi_{m-1} y_{N-2} + \dots + \phi_0 y_{N-m-1} + \phi_1 y_{N-m-2} + \\ \dots + \phi_m y_{N-2m-1} = a_{N-m-1} \\ \phi_m y_{N-2} + \phi_{m-1} y_{N-3} + \dots + \phi_0 y_{N-m-2} + \phi_1 y_{N-m-3} + \\ \dots + \phi_m y_{N-2m-2} = a_{N-m-2} \\ \vdots \\ \phi_m y_{m+1} + \phi_{m-1} y_m + + \dots + \phi_0 y_1 + \phi_1 y_0 + \phi_m y_{-m+1} = a_1 \\ \phi_m y_m + \phi_{m-1} y_{m-1} + \phi_{m-2} + \dots + \phi_0 y_0 + \phi_1 y_{-1} + \dots + \phi_m y_{-m} = a_0 \end{aligned}$$

As before, we can express this equation as $W\bar{y} = \bar{a}$

The matrix on the left of this equation is "almost" Toeplitz, the exception being the leading $m \times m$ 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

$$U\bar{y} = L^{-1}\bar{a} \tag{4.140}$$

$$\sum_{j=0}^{t} U_{-t+N+1,\,-t+N+j+1}\, y_{t-j} = \sum_{j=0}^{N-t} L_{-t+N+1,\,-t+N+1-j}\, \bar{a}_{t+j}\ ,$$
$$t = 0, 1, \ldots, N$$

where $L^{-1}_{t,s}$ is the element in the (t, s) position of L, and similarly for U

The left side of equation (4.140) 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 $N \to \infty$, (4.140) becomes equivalent to the solution of our infinite horizon problem, which below we shall show can be expressed as

$$c(L)y_t = c(\beta L^{-1})^{-1} a_t\ ,$$

so that as $N \to \infty$ we expect that for each fixed $t, U^{-1}_{t,t-j} \to c_j$ and $L_{t,t+j}$ approaches the coefficient on L^{-j} in the expansion of $c(\beta L^{-1})^{-1}$

This suspicion is true under general conditions that we shall study later

For now, we note that by creating the matrix W for large N and factoring it into the LU form, good approximations to $c(L)$ and $c(\beta L^{-1})^{-1}$ can be obtained

The Infinite Horizon Limit

For the infinite horizon problem, we propose to discover first-order necessary conditions by taking the limits of (4.133) and (4.134) as $N \to \infty$

This approach is valid, and the limits of (4.133) and (4.134) as N approaches infinity are first-order necessary conditions for a maximum

However, for the infinite horizon problem with $\beta < 1$, the limits of (4.133) and (4.134) are, in general, not sufficient for a maximum

That is, the limits of (4.134) do not provide enough information uniquely to determine the solution of the Euler equation (4.133) that maximizes (4.130)

As we shall see below, a side condition on the path of y_t that together with (4.133) is sufficient for an optimum is

$$\sum_{t=0}^{\infty} \beta^t\, h y_t^2 < \infty \tag{4.141}$$

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 (4.130) than does the optimal path selected by the solution procedure below

Consider the *characteristic equation* associated with the Euler equation

$$h + d\left(\beta z^{-1}\right) d\left(z\right) = 0 \tag{4.142}$$

Notice that if $\tilde{z}$ is a root of equation (4.142), then so is $\beta \tilde{z}^{-1}$

Thus, the roots of (4.142) come in "β-reciprocal" pairs

Assume that the roots of (4.142) 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 (4.142) can be expressed as

$$\begin{aligned} h + d(\beta z^{-1})d(z) &= z^{-m} z_0 (z - z_1) \cdots (z - z_m)(z - z_{m+1}) \cdots (z - z_{2m}) \\ &= z^{-m} z_0 (z - z_1)(z - z_2) \cdots (z - z_m)(z - \beta z_m^{-1}) \cdots (z - \beta z_2^{-1})(z - \beta z_1^{-1}) \end{aligned} \tag{4.143}$$

where z_0 is a constant

In (4.143), we substitute $(z - z_j) = -z_j(1 - \frac{1}{z_j} z)$ and $(z - \beta z_j^{-1}) = z(1 - \frac{\beta}{z_j} z^{-1})$ for $j = 1, \ldots, m$ to get

$$h + d(\beta z^{-1})d(z) = (-1)^m (z_0 z_1 \cdots z_m)(1 - \frac{1}{z_1} z) \cdots (1 - \frac{1}{z_m} z)(1 - \frac{1}{z_1} \beta z^{-1}) \cdots (1 - \frac{1}{z_m} \beta z^{-1})$$

Now define $c(z) = \sum_{j=0}^{m} c_j \, z^j$ as

$$c\left(z\right) = \left[(-1)^m z_0 \, z_1 \cdots z_m\right]^{1/2} (1 - \frac{z}{z_1}) \, (1 - \frac{z}{z_2}) \cdots (1 - \frac{z}{z_m}) \tag{4.144}$$

Notice that (4.143) can be written

$$h + d\left(\beta z^{-1}\right) d\left(z\right) = c\left(\beta z^{-1}\right) c\left(z\right) \tag{4.145}$$

It is useful to write (4.144) as

$$c(z) = c_0 (1 - \lambda_1 \, z) \ldots (1 - \lambda_m z) \tag{4.146}$$

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 (4.146), we can express the factorization (4.145) as

$$h + d(\beta z^{-1})d(z) = c_0^2(1-\lambda_1 z)\cdots(1-\lambda_m z)(1-\lambda_1\beta z^{-1})\cdots(1-\lambda_m\beta z^{-1})$$

In sum, we have constructed a factorization (4.145) of the characteristic polynominal for the Euler equation in which the zeros of $c(z)$ exceed $\beta^{1/2}$ in modulus, and the zeros of $c\left(\beta z^{-1}\right)$ are less than $\beta^{1/2}$ in modulus

Using (4.145), we now write the Euler equation as

$$c(\beta L^{-1})\, c\left(L\right) y_t = a_t$$

The unique solution of the Euler equation that satisfies condition (4.141) is

$$c(L)\, y_t = c\left(\beta L^{-1}\right)^{-1} a_t \tag{4.147}$$

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 (4.146) to write (4.147) as

$$(1-\lambda_1 L)\cdots(1-\lambda_m L)y_t = \frac{c_0^{-2} a_t}{(1-\beta\lambda_1 L^{-1})\cdots(1-\beta\lambda_m L^{-1})} \tag{4.148}$$

Using partial fractions, we can write the characteristic polynomial on the right side of (4.148) 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 (4.148) 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

$$(1-\lambda_1 L)\cdots(1-\lambda_m L)y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} \left(\lambda_j \beta\right)^k a_{t+k} \tag{4.149}$$

Equation (4.149) expresses the optimum sequence for y_t in terms of m lagged y's, and m weighted infinite geometric sums of future a_t's

Furthermore, (4.149) is the unique solution of the Euler equation that satisfies the initial conditions and condition (4.141)

In effect, condition (4.141) 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 $c\left(\beta z^{-1}\right)c\left(z\right)$, where the zeros of $c\left(z\right)$ all have modulus exceeding $\sqrt{\beta}$, is central to solving the problem

We note two features of the solution (4.149)

- Since $|\lambda_j| < 1/\sqrt{\beta}$ for all j, it follows that $(\lambda_j\ \beta) < \sqrt{\beta}$

- The assumption that $\{a_t\}$ is of exponential order less than $1/\sqrt{\beta}$ is sufficient to guarantee that the geometric sums of future a_t's on the right side of (4.149) converge

We immediately see that those sums will converge under the weaker condition that $\{a_t\}$ is of exponential order less than ϕ^{-1} where $\phi = \max\{\beta\lambda_i, i = 1, \ldots, m\}$

Note that with a_t identically zero, (4.149) 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 (4.141) is satisfied

In fact, $\max_i |\lambda_i| < 1/\sqrt{\beta}$ is a necessary condition for (4.141) to hold

Were (4.141) 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 (4.146)

We can simply let $y_t = 0$ for $t \geq 0$

This policy involves at most m nonzero values of hy_t^2 and $[d(L)y_t]^2$, and so yields a finite value of (4.130)

Therefore it is easy to dominate a path that violates (4.141)

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 (4.145) becomes

$$\left(h + d\left(z^{-1}\right)d(z)\right) = c\left(z^{-1}\right)c\left(z\right)$$

where

$$\begin{aligned}
c\left(z\right) &= c_0(1 - \lambda_1 z)\ldots(1 - \lambda_m z) \\
c_0 &= \left[(-1)^m z_0 z_1 \ldots z_m\right] \\
|\lambda_j| &< 1 \text{ for } j = 1, \ldots, m \\
\lambda_j &= \frac{1}{z_j} \text{ for } j = 1, \ldots, m \\
z_0 &= \text{ constant}
\end{aligned}$$

The solution of the problem becomes

$$(1 - \lambda_1 L)\cdots(1 - \lambda_m L)y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} \lambda_j^k a_{t+k}$$

Transforming discounted to undiscounted problem

Discounted problems can always be converted into undiscounted problems via a simple transformation

Consider problem (4.130) with $0 < \beta < 1$

Define the transformed variables

$$\tilde{a}_t = \beta^{t/2} a_t,\ \tilde{y}_t = \beta^{t/2} y_t \tag{4.150}$$

Then notice that $\beta^t \left[d\left(L\right) y_t\right]^2 = [\tilde{d}\left(L\right)\tilde{y}_t]^2$ with $\tilde{d}\left(L\right) = \sum_{j=0}^m \tilde{d}_j\, L^j$ and $\tilde{d}_j = \beta^{j/2} d_j$

Then the original criterion function (4.130) is equivalent to

$$\lim_{N\to\infty} \sum_{t=0}^N \{\tilde{a}_t\, \tilde{y}_t - \frac{1}{2} h\, \tilde{y}_t^2 - \frac{1}{2}[\tilde{d}\left(L\right)\tilde{y}_t]^2\} \tag{4.151}$$

which is to be maximized over sequences $\{\tilde{y}_t,\ t = 0, \ldots\}$ subject to $\tilde{y}_{-1}, \cdots, \tilde{y}_{-m}$ given and $\{\tilde{a}_t,\ t = 1, \ldots\}$ a known bounded sequence

The Euler equation for this problem is $[h + \tilde{d}\left(L^{-1}\right)\tilde{d}\left(L\right)]\, \tilde{y}_t = \tilde{a}_t$

The solution is

$$(1 - \tilde{\lambda}_1 L) \cdots (1 - \tilde{\lambda}_m L)\, \tilde{y}_t = \sum_{j=1}^m \tilde{A}_j \sum_{k=0}^\infty \tilde{\lambda}_j^k\, \tilde{a}_{t+k}$$

or

$$\tilde{y}_t = \tilde{f}_1\, \tilde{y}_{t-1} + \cdots + \tilde{f}_m\, \tilde{y}_{t-m} + \sum_{j=1}^m \tilde{A}_j \sum_{k=0}^\infty \tilde{\lambda}_j^k\, \tilde{a}_{t+k}, \tag{4.152}$$

where $\tilde{c}\left(z^{-1}\right)\tilde{c}\left(z\right) = h + \tilde{d}\left(z^{-1}\right)\tilde{d}\left(z\right)$, and where

$$\left[(-1)^m\, \tilde{z}_0 \tilde{z}_1 \ldots \tilde{z}_m\right]^{1/2} (1 - \tilde{\lambda}_1\, z) \ldots (1 - \tilde{\lambda}_m\, z) = \tilde{c}\left(z\right),\ \text{where}\ |\tilde{\lambda}_j| < 1$$

We leave it to the reader to show that (4.152) implies the equivalent form of the solution

$$y_t = f_1\, y_{t-1} + \cdots + f_m\, y_{t-m} + \sum_{j=1}^m A_j \sum_{k=0}^\infty (\lambda_j\, \beta)^k\, a_{t+k}$$

where

$$f_j = \tilde{f}_j\, \beta^{-j/2},\ A_j = \tilde{A}_j,\ \lambda_j = \tilde{\lambda}_j\, \beta^{-1/2} \tag{4.153}$$

The transformations (4.150) and the inverse formulas (4.153) allow us to solve a discounted problem by first solving a related undiscounted problem

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

```
"""
Classical discrete time LQ optimal control and filtering problems. The
control problems we consider take the form

    \max \sum_{t = 0}^N \beta^t \{a_t y_t - h y^2_t / 2 - [d(L)y_t]^2 \}

subject to h > 0, 0 < \beta < 1 and

 * y_{-1},y_{-2}, \dots, y_{-m} (initial conditions)

 * d(L) = d_0 + d_1L + d_2L^2 + \dots + d_mL^m

The sequence {y_t} is scalar

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
                    Parameter of the objective function (corresponding to the
                    quadratic term)
            y_m : list or numpy.array (1-D or a 2-D column vector)
                    Initial conditions for y
            r : list or numpy.array (1-D or a 2-D column vector)
                    The order of the coefficients: [r_0, r_1, ..., r_k]
                    (optional, if not defined -> deterministic problem)
            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 = {:d}".format(self.m))

        #-------------------------------------------
        # Define the coefficients of phi up front
        #-------------------------------------------
        phi = np.zeros(2 * self.m + 1)
        for i in range(- self.m, self.m + 1):
            phi[self.m - i] = np.sum(np.diag(self.d.reshape(self.m + 1, 1) @ \
                                             self.d.reshape(1, self.m +␣
↪1), k = -i))
        phi[self.m] = phi[self.m] + self.h
        self.phi = phi

        #-----------------------------------------------------
        # If r is given calculate the vector phi_r
        #-----------------------------------------------------
        if r is None:
            pass
        else:
            self.r = np.asarray(r)
            self.k = self.r.shape[0] - 1
            phi_r = np.zeros(2 * self.k + 1)
            for i in range(- self.k, self.k + 1):
                phi_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.phi_r = phi_r
            else:
                phi_r[self.k] = phi_r[self.k] + h_eps
                self.phi_r = phi_r

        #-----------------------------------------------------
        # If beta is given, define the transformed variables
        #-----------------------------------------------------
        if beta is None:
            self.beta = 1
        else:
            self.beta = beta
            self.d = self.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) Constuct the D_{m+1} matrix using the formula

        for j in range(m + 1):
            for k in range(j, m + 1):
                D_m1[j, k] = d[:j + 1] @ d[k - j : k + 1]

        # Make the matrix symmetric
        D_m1 = D_m1 + D_m1.T - np.diag(np.diag(D_m1))

        # (2) Construct the M matrix using the entries of D_m1

        for j in range(m):
            for i in range(j + 1, m + 1):
                M[i, j] = D_m1[i - j - 1, m]

        #----------------------------------------------
        # Euler equations for t = 0, 1, ..., N-(m+1)
        #----------------------------------------------
        phi = self.phi

        W[:(m + 1), :(m + 1)] = D_m1 + self.h * np.eye(m + 1)
        W[:(m + 1), (m + 1):(2 * m + 1)] = M

        for i, row in enumerate(np.arange(m + 1, N + 1 - m)):
            W[row, (i + 1):(2 * m + 2 + i)] = phi

        for i in range(1, m + 1):
            W[N - m + i, -(2 * m + 1 - i):] = phi[:-i]

        for i in range(m):
            W_m[N - i, :(m - i)] = phi[(m + 1 + i):]

        return W, W_m



    def roots_of_characteristic(self):
        """
        This function calculates z_0 and the 2m roots of the characteristic␣
↪equation
        associated with the Euler equation (1.7)

        Note:
```

```
        ------
        numpy.poly1d(roots, True) defines a polynomial using its roots that␣
↪can be
        evaluated at any point. If x_1, x_2, ... , x_m are the roots then
            p(x) = (x - x_1)(x - x_2)...(x - x_m)

        """
        m = self.m
        phi = self.phi

        # Calculate the roots of the 2m-polynomial
        roots = np.roots(phi)
        # sort the roots according to their length (in descending order)
        roots_sorted = roots[np.argsort(abs(roots))[::-1]]

        z_0 = phi.sum() / np.poly1d(roots, True)(1)
        z_1_to_m = roots_sorted[:m]     # we need only those outside the unit␣
↪circle

        lambdas = 1 / z_1_to_m

        return z_1_to_m, z_0, lambdas


    def coeffs_of_c(self):
        '''
        This function computes the coefficients {c_j, j = 0, 1, ..., m} for
                c(z) = sum_{j = 0}^{m} c_j z^j

        Based on the expression (1.9). The order is
            c_coeffs = [c_0, c_1, ..., c_{m-1}, c_m]
        '''
        z_1_to_m, z_0 = self.roots_of_characteristic()[:2]

        c_0 = (z_0 * np.prod(z_1_to_m).real * (- 1)**self.m)**(.5)
        c_coeffs = np.poly1d(z_1_to_m, True).c * z_0 / c_0

        return c_coeffs[::-1]


    def solution(self):
        """
        This function calculates {lambda_j, j=1,...,m} and {A_j, j=1,...,m}
        of the expression (1.15)
        """
        lambdas = 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 - lambdas/lambdas[j]
            A[j] = c_0**(-2) / np.prod(denom[np.arange(self.m) != j])
```

```
        return lambdas, 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))
        phi_r = self.phi_r

        for i in range(N):
            for j in range(N):
                if abs(i-j) <= self.k:
                    V[i, j] = phi_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 is section␣
↪6 (1.59)
        It takes a realization for a^N, and the period in which the␣
↪prediciton is formed

        Output:  E[abar | a_t, a_{t-1}, ..., a_1, a_0]
        """

        N = np.asarray(a_hist).shape[0] - 1
        a_hist = np.asarray(a_hist).reshape(N + 1, 1)
        V = self.construct_V(N + 1)

        aux_matrix = np.zeros((N + 1, N + 1))
        aux_matrix[:(t + 1), :(t + 1)] = np.eye(t + 1)
        L = la.cholesky(V).T
        Ea_hist = la.inv(L) @ aux_matrix @ L @ a_hist

        return Ea_hist


    def optimal_y(self, a_hist, t = None):
```

```
        """
        - if t is NOT given it takes a_hist (list or numpy.array) as a␣
↪deterministic a_t
        - if t is given, it solves the combined control prediction problem␣
↪(section 7)
          (by default, t == None -> deterministic)

        for a given sequence of a_t (either 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 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 the lecutre
            Uy = np.linalg.solve(L, a_bar)            # U @ y_bar = L^{-1}a_
↪bar from the lecture
            y_bar = np.linalg.solve(U, Uy)            # y_bar = U^{-1}L^{-1}a_
↪bar

            # 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␣
↪lecutre
            Uy = np.linalg.solve(L, a_bar)            # U @ y_bar = L^{-1}a_
↪bar from the lecture
            y_bar = np.linalg.solve(U, Uy)            # y_bar = U^{-1}L^{-1}a_
↪bar

            # 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'll have one lag, with

$$d(L)y_t = \gamma(I - L)y_t = \gamma(y_t - y_{t-1})$$

Suppose for the moment that $\gamma = 0$

Then the intertemporal component of the LQ problem disappears, and the agent simply wants to maximize $a_t y_t - h y_t^2/2$ in each period

This means that the agent chooses $y_t = a_t/h$

In the following we'll set $h = 1$, so that the agent just wants to track the $\{a_t\}$ process

However, as we increase γ, the agent gives greater weight to a smooth time path

Hence $\{y_t\}$ evolves as a smoothed version of $\{a_t\}$

The $\{a_t\}$ sequence we'll choose as a stationary cyclic process plus some white noise

Here's some code that generates a plot when $\gamma = 0.8$

```
import numpy as np
```

```
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)

# == Model parameters == #
gamma = 0.8
m = 1
d = gamma * np.asarray([1, -1])
h = 1.0

# == Initial conditions == #
y_m = np.asarray([2.0]).reshape(m, 1)

testlq = LQFilter(d, h, y_m)
y_hist, L, U, y = testlq.optimal_y(a_seq)
y = y[::-1]  # reverse y

# == Plot simulation results == #
fig, ax = plt.subplots(figsize=(10, 6.1))
ax.set_xlabel('Time')

# == Some fancy plotting stuff -- simplify if you prefer == #
bbox = (0., 1.01, 1., .101)
legend_args = {'bbox_to_anchor' : bbox, 'loc' : 3, 'mode' : 'expand',
↪'fontsize': 16}
p_args = {'lw' : 2, 'alpha' : 0.6}

time = range(len(y))
ax.set_xlim(0, max(time))
ax.plot(time, a_seq / h, 'k-o', ms=4, lw=2, alpha=0.6, label=r'$a_t$')
ax.plot(time, y, 'b-o', ms=4, lw=2, alpha=0.6, label=r'$y_t$')
ax.legend(ncol=2, **legend_args)
ax.grid()
s = r'dynamics with $\gamma = {}$'.format(gamma)
plt.show()
```

Here's the figure that it generates

Here's what happens when we change γ to 5.0

And here's $\gamma = 10$

Exercises

Exercise 1

Consider solving a discounted version $(\beta < 1)$ of problem (4.130), as follows

Convert (4.130) to the undiscounted problem (4.151)

Let the solution of (4.151) in feedback form be

$$(1 - \tilde{\lambda}_1 L) \cdots (1 - \tilde{\lambda}_m L)\tilde{y}_t = \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k}$$

or

$$\tilde{y}_t = \tilde{f}_1 \tilde{y}_{t-1} + \cdots + \tilde{f}_m \tilde{y}_{t-m} + \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k} \tag{4.154}$$

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 (4.154) implies that the solution for y_t in feedback form is

$$y_t = f_1 y_{t-1} + \ldots + f_m y_{t-m} + \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} \beta^k \lambda_j^k a_{t+k}$$

where $f_j = \tilde{f}_j \beta^{-j/2}$, $A_j = \tilde{A}_j$, and $\lambda_j = \tilde{\lambda}_j \beta^{-1/2}$

Exercise 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" (4.149), giving numerical values for the coefficients

Make sure that the boundary conditions (4.134) are satisfied

(Note: this problem differs from the problem in the text in one important way: instead of $h > 0$ in (4.130), $h = 0$. This has an important influence on the solution.)

Exercise 3

Solve the infinite time optimal control problem to maximize

$$\lim_{N \to \infty} \sum_{t=0}^{N} -\frac{1}{2}[(1-2L)y_t]^2,$$

subject to y_{-1} given. Prove that the solution is

$$y_t = 2y_{t-1} = 2^{t+1} y_{-1} \qquad t > 0$$

Exercise 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 (4.141), and so is not optimal

Prove that the optimal solution is approximately $y_t = .5y_{t-1}$

Contents

- *Classical Filtering With Linear Algebra*
  - *Overview*
  - *Infinite Horizon Prediction and Filtering Problems*
  - *Finite Dimensional Prediction*

– *Combined Finite Dimensional Control and Prediction*
– *Exercises*

Classical Filtering With Linear Algebra

Overview

This is a sequel to the earlier lecture *Classical Control with Linear Algebra*

That lecture used linear algebra – in particular, the LU decomposition – to formulate and solve a class of linear-quadratic optimal control problems

In this lecture, we'll be using a closely related decomposition, the Cholesky decomposition , to solve linear prediction and filtering problems

We exploit the useful fact that there is an intimate connection between two superficially different classes of problems:

- deterministic linear-quadratic (LQ) optimal control problems
- linear least squares prediction and filtering problems

The first class of problems involves no randomness, while the second is all about randomness

Nevertheless, essentially the same mathematics solves both 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]*

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,

$$Y_t = d(L)u_t \tag{4.155}$$

where $d(L) = \sum_{j=0}^{m} d_j L^j$, and u_t is a serially uncorrelated stationary random process satisfying

$$\begin{aligned} \mathbb{E}u_t &= 0 \\ \mathbb{E}u_t u_s &= \begin{cases} 1 & \text{if } t = s \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{4.156}$$

We impose no conditions on the zeros of $d(z)$

A second covariance stationary process is X_t given by

$$X_t = Y_t + \varepsilon_t \tag{4.157}$$

where ε_t is a serially uncorrelated stationary random process with $\mathbb{E}\varepsilon_t = 0$ and $\mathbb{E}\varepsilon_t\varepsilon_s$ = 0 for all distinct t and s

We also assume that $\mathbb{E}\varepsilon_t u_s = 0$ for all t and s

The **linear least squares prediction problem** is to find the L_2 random variable $\hat{X}_{t+j}$ among linear combinations of $\{X_t,\ X_{t-1}, \ldots\}$ 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,

$$\begin{aligned} C_X(\tau) &= \mathbb{E}X_t X_{t-\tau} \\ C_Y(\tau) &= \mathbb{E}Y_t Y_{t-\tau} \qquad \tau = 0, \pm 1, \pm 2, \ldots \\ C_{Y,X}(\tau) &= \mathbb{E}Y_t X_{t-\tau} \end{aligned} \tag{4.158}$$

The covariance and cross covariance generating functions are defined as

$$\begin{aligned} g_X(z) &= \sum_{\tau=-\infty}^{\infty} C_X(\tau) z^\tau \\ g_Y(z) &= \sum_{\tau=-\infty}^{\infty} C_Y(\tau) z^\tau \\ g_{YX}(z) &= \sum_{\tau=-\infty}^{\infty} C_{YX}(\tau) z^\tau \end{aligned} \tag{4.159}$$

The generating functions can be computed by using the following facts

Let v_{1t} and v_{2t} be two mutually and serially uncorrelated white noises with unit variances

That is, $\mathbb{E}v_{1t}^2 = \mathbb{E}v_{2t}^2 = 1, \mathbb{E}v_{1t} = \mathbb{E}v_{2t} = 0, \mathbb{E}v_{1t}v_{2s} = 0$ for all t and s, $\mathbb{E}v_{1t}v_{1t-j} = \mathbb{E}v_{2t}v_{2t-j} = 0$ for all $j \neq 0$

Let x_t and y_t be two random process given by

$$\begin{aligned} y_t &= A(L)v_{1t} + B(L)v_{2t} \\ x_t &= C(L)v_{1t} + D(L)v_{2t} \end{aligned}$$

Then, as shown for example in *[Sar87]* [ch. XI], it is true that

$$\begin{aligned} g_y(z) &= A(z)A(z^{-1}) + B(z)B(z^{-1}) \\ g_x(z) &= C(z)C(z^{-1}) + D(z)D(z^{-1}) \\ g_{yx}(z) &= A(z)C(z^{-1}) + B(z)D(z^{-1}) \end{aligned} \tag{4.160}$$

Applying these formulas to (4.155) – (4.158), we have

$$\begin{aligned} g_Y(z) &= d(z)d(z^{-1}) \\ g_X(z) &= d(z)d(z^{-1}) + h \\ g_{YX}(z) &= d(z)d(z^{-1}) \end{aligned} \tag{4.161}$$

The key step in obtaining solutions to our problems is to factor the covariance generating function $g_X(z)$ of X

The solutions of our problems are given by formulas due to Wiener and Kolmogorov

These formulas utilize the Wold moving average representation of the X_t process,

$$X_t = c\left(L\right)\eta_t \tag{4.162}$$

where $c(L) = \sum_{j=0}^{m} c_j\, L^j$, with

$$c_0\eta_t = X_t - \hat{\mathbb{E}}[X_t|X_{t-1}, X_{t-2}, \ldots] \tag{4.163}$$

Here $\hat{\mathbb{E}}$ is the linear least squares projection operator

Equation (4.163) 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 (4.163) requires that η_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 (4.163) that η_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 (4.162) is that the covariance generating function of X_t can be expressed as

$$g_X(z) = c\,(z)\,c\,(z^{-1}) \tag{4.164}$$

It remains to discuss how $c(L)$ is to be computed

Combining (4.160) and (4.164) gives

$$d(z)\,d(z^{-1}) + h = c\,(z)\,c\,(z^{-1}) \tag{4.165}$$

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**:

$$\left[\sum_{j=-\infty}^{\infty} f_j\,L^j\right]_+ \equiv \sum_{j=0}^{\infty} f_j\,L^j \tag{4.166}$$

In words, $[\quad]_+$ means "ignore negative powers of L"

We have defined the solution of the prediction problem as $\hat{\mathbb{E}}[X_{t+j}|X_t,\ X_{t-1}, \ldots] = \gamma_j\,(L)X_t$

Assuming that the roots of $c(z) = 0$ all lie outside the unit circle, the Wiener-Kolmogorov formula for $\gamma_j(L)$ holds:

$$\gamma_j\,(L) = \left[\frac{c(L)}{L^j}\right]_+ c\,(L)^{-1} \tag{4.167}$$

We have defined the solution of the filtering problem as $\hat{\mathbb{E}}[Y_t \mid X_t, X_{t-1}, \ldots] = b(L)X_t$

The Wiener-Kolomogorov formula for $b(L)$ is

$$b(L) = \left(\frac{g_{YX}(L)}{c(L^{-1})}\right)_+ c(L)^{-1}$$

or

$$b(L) = \left[\frac{d(L)d(L^{-1})}{c(L^{-1})}\right]_+ c(L)^{-1} \tag{4.168}$$

Formulas (4.167) and (4.168) 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 (4.168), we let X_t be a stochastic process with Wold moving average representation

$$X_t = c(L)\eta_t$$

where $\mathbb{E}\eta_t^2 = 1,$ and $c_0\eta_t = X_t - \hat{\mathbb{E}}[X_t|X_{t-1},\ldots], c(L) = \sum_{j=0}^m c_j L$

Suppose that at time t, we wish to predict a geometric sum of future X's, namely

$$y_t \equiv \sum_{j=0}^{\infty} \delta^j X_{t+j} = \frac{1}{1-\delta L^{-1}} X_t$$

given knowledge of $X_t, X_{t-1}, \ldots$

We shall use (4.168) to obtain the answer

Using the standard formulas (4.160), we have that

$$\begin{aligned} g_{yx}(z) &= (1-\delta z^{-1})c(z)c(z^{-1}) \\ g_x(z) &= c(z)c(z^{-1}) \end{aligned}$$

Then (4.168) becomes

$$b(L) = \left[\frac{c(L)}{1-\delta L^{-1}}\right]_+ c(L)^{-1} \tag{4.169}$$

In order to evaluate the term in the annihilation operator, we use the following result from *[HS80]*

Proposition Let

- $g(z) = \sum_{j=0}^{\infty} g_j\, z^j$ where $\sum_{j=0}^{\infty} |g_j|^2 < +\infty$
- $h\left(z^{-1}\right) = (1-\delta_1 z^{-1})\ldots(1-\delta_n z^{-1})$, where $|\delta_j| < 1$, for $j = 1, \ldots, n$

Then

$$\left[\frac{g(z)}{h(z^{-1})}\right]_+ = \frac{g(z)}{h(z^{-1})} - \sum_{j=1}^{n} \frac{\delta_j g(\delta_j)}{\prod_{\substack{k=1\\k\neq j}}^{n}(\delta_j - \delta_k)} \left(\frac{1}{z-\delta_j}\right) \tag{4.170}$$

and, alternatively,

$$\left[\frac{g(z)}{h(z^{-1})}\right]_+ = \sum_{j=1}^{n} B_j \left(\frac{zg(z) - \delta_j g(\delta_j)}{z-\delta_j}\right) \tag{4.171}$$

where $B_j = 1/\prod_{\substack{k=1\\k+j}}^{n}(1-\delta_k/\delta_j)$

Applying formula (4.171) of the proposition to evaluating (4.169) with $g(z) = c(z)$ and $h(z^{-1}) = 1-\delta z^{-1}$ gives

$$b(L) = \left[\frac{Lc(L) - \delta c(\delta)}{L-\delta}\right] c(L)^{-1}$$

or

$$b(L) = \left[\frac{1-\delta c(\delta)L^{-1}c(L)^{-1}}{1-\delta L^{-1}}\right]$$

Thus, we have

$$\hat{\mathbb{E}}\left[\sum_{j=0}^{\infty} \delta^j X_{t+j}|X_t,\, x_{t-1},\ldots\right] = \left[\frac{1-\delta c(\delta)L^{-1}c(L)^{-1}}{1-\delta L^{-1}}\right] X_t \tag{4.172}$$

This formula is useful in solving stochastic versions of problem 1 of lecture *Classical Control with Linear Algebra* in which the randomness emerges because $\{a_t\}$ is a stochastic process

The problem is to maximize

$$\mathbb{E}_0 \lim_{N\to\infty} \sum_{t-0}^{N} \beta^t \left[a_t\, y_t - \frac{1}{2}\, h y_t^2 - \frac{1}{2}\, [d(L) y_t]^2 \right] \tag{4.173}$$

where $\mathbb{E}_t$ is mathematical expectation conditioned on information known at t, and where $\{a_t\}$ is a covariance stationary stochastic process with Wold moving average representation

$$a_t = c(L)\, \eta_t$$

where

$$c(L) = \sum_{j=0}^{\tilde{n}} c_j L^j$$

and

$$\eta_t = a_t - \hat{\mathbb{E}}[a_t | a_{t-1}, \ldots]$$

The problem is to maximize (4.173) with respect to a contingency plan expressing y_t as a function of information known at t, which is assumed to be $(y_{t-1},\ y_{t-2}, \ldots, a_t,\ a_{t-1}, \ldots)$

The solution of this problem can be achieved in two steps

First, ignoring the uncertainty, we can solve the problem assuming that $\{a_t\}$ is a known sequence

The solution is, from above,

$$c(L) y_t = c(\beta L^{-1})^{-1} a_t$$

or

$$(1 - \lambda_1 L) \ldots (1 - \lambda_m L) y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} (\lambda_j \beta)^k\, a_{t+k} \tag{4.174}$$

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 (4.172) and (4.174), we have the following solution

$$(1 - \lambda_1 L) \ldots (1 - \lambda_m L) y_t = \sum_{j=1}^{m} A_j \left[\frac{1 - \beta \lambda_j\, c(\beta \lambda_j) L^{-1} c(L)^{-1}}{1 - \beta \lambda_j L^{-1}} \right] a_t$$

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 (4.162)

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

$$\hat{\mathbb{E}}\left[x_T | x_{T-j}, x_{T-j+1}, \ldots, x_1\right]$$

where $\hat{\mathbb{E}}$ is the linear least squares projection operator

The solution of this problem can be exhibited by first constructing an orthonormal basis of random variables ε 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

$$L\, V\, L' = I$$

where L is lower-trangular, and therefore so is L^{-1}

Form the random variable $Lx = \varepsilon$

Then ε is an orthonormal basis for x, since L is nonsingular, and $\mathbb{E}\, \varepsilon\, \varepsilon' = L \mathbb{E} x x' L' = I$

It is convenient to write out the equations $Lx = \varepsilon$ and $L^{-1}\varepsilon = x$

$$\begin{aligned} L_{11} x_1 &= \varepsilon_1 \\ L_{21} x_1 + L_{22} x_2 &= \varepsilon_2 \\ &\vdots \\ L_{T1}\, x_1 \ \ldots + L_{TT x_T} &= \varepsilon_T \end{aligned} \tag{4.175}$$

or

$$\sum_{j=0}^{t-1} L_{t,t-j}\, x_{t-j} = \varepsilon_t, \quad t = 1,\, 2, \ldots T \tag{4.176}$$

We also have

$$x_t = \sum_{j=0}^{t-1} L^{-1}_{t,t-j}\, \varepsilon_{t-j}\ . \tag{4.177}$$

Notice from (4.177) that x_t is in the space spanned by $\varepsilon_t,\, \varepsilon_{t-1}, \ldots, \varepsilon_1$, and from (4.176) that ε_t is in the space spanned by $x_t,\, x_{t-1}, \ldots,\, x_1$

Therefore, we have that for $t-1 \geq m \geq 1$

$$\hat{\mathbb{E}}[x_t \mid x_{t-m}, x_{t-m-1}, \ldots, x_1] = \hat{\mathbb{E}}[x_t \mid \varepsilon_{t-m}, \varepsilon_{t-m-1}, \ldots, \varepsilon_1] \tag{4.178}$$

For $t-1 \geq m \geq 1$ rewrite (4.177) as

$$x_t = \sum_{j=0}^{m-1} L_{t,t-j}^{-1} \varepsilon_{t-j} + \sum_{j=m}^{t-1} L_{t,t-j}^{-1} \varepsilon_{t-j} \tag{4.179}$$

Representation (4.179) 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

```
"""
Classical discrete time LQ optimal control and filtering problems. The
control problems we consider take the form

    \max \sum_{t = 0}^N \beta^t \{a_t y_t - h y^2_t / 2 - [d(L)y_t]^2 \}

subject to h > 0, 0 < \beta < 1 and

 * y_{-1},y_{-2}, \dots, y_{-m} (initial conditions)

 * d(L) = d_0 + d_1L + d_2L^2 + \dots + d_mL^m

The sequence {y_t} is scalar

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
                    Parameter of the objective function (corresponding to the
```

```
                quadratic term)
        y_m : list or numpy.array (1-D or a 2-D column vector)
                Initial conditions for y
        r : list or numpy.array (1-D or a 2-D column vector)
                The order of the coefficients: [r_0, r_1, ..., r_k]
                (optional, if not defined -> deterministic problem)
        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 = {:d}".format(self.m))

    #---------------------------------------------
    # Define the coefficients of phi up front
    #---------------------------------------------
    phi = np.zeros(2 * self.m + 1)
    for i in range(- self.m, self.m + 1):
        phi[self.m - i] = np.sum(np.diag(self.d.reshape(self.m + 1, 1) @ \
                                         self.d.reshape(1, self.m +␣
↪1), k = -i))
    phi[self.m] = phi[self.m] + self.h
    self.phi = phi

    #-----------------------------------------------------
    # If r is given calculate the vector phi_r
    #-----------------------------------------------------
    if r is None:
        pass
    else:
        self.r = np.asarray(r)
        self.k = self.r.shape[0] - 1
        phi_r = np.zeros(2 * self.k + 1)
        for i in range(- self.k, self.k + 1):
            phi_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.phi_r = phi_r
        else:
            phi_r[self.k] = phi_r[self.k] + h_eps
            self.phi_r = phi_r
```

```
        #-----------------------------------------------------
        # If beta is given, define the transformed variables
        #-----------------------------------------------------
        if beta is None:
            self.beta = 1
        else:
            self.beta = beta
            self.d = self.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) Constuct the D_{m+1} matrix using the formula

        for j in range(m + 1):
            for k in range(j, m + 1):
                D_m1[j, k] = d[:j + 1] @ d[k - j : k + 1]

        # Make the matrix symmetric
        D_m1 = D_m1 + D_m1.T - np.diag(np.diag(D_m1))

        # (2) Construct the M matrix using the entries of D_m1

        for j in range(m):
            for i in range(j + 1, m + 1):
                M[i, j] = D_m1[i - j - 1, m]

        #-----------------------------------------------
        # Euler equations for t = 0, 1, ..., N-(m+1)
        #-----------------------------------------------
        phi = self.phi

        W[:(m + 1), :(m + 1)] = D_m1 + self.h * np.eye(m + 1)
        W[:(m + 1), (m + 1):(2 * m + 1)] = M
```

```
        for i, row in enumerate(np.arange(m + 1, N + 1 - m)):
            W[row, (i + 1):(2 * m + 2 + i)] = phi

        for i in range(1, m + 1):
            W[N - m + i, -(2 * m + 1 - i):] = phi[:-i]

        for i in range(m):
            W_m[N - i, :(m - i)] = phi[(m + 1 + i):]

        return W, W_m



    def roots_of_characteristic(self):
        """
        This function calculates z_0 and the 2m roots of the characteristic␣
↪equation
        associated with the Euler equation (1.7)

        Note:
        ------
        numpy.poly1d(roots, True) defines a polynomial using its roots that␣
↪can be
        evaluated at any point. If x_1, x_2, ... , x_m are the roots then
            p(x) = (x - x_1)(x - x_2)...(x - x_m)

        """
        m = self.m
        phi = self.phi

        # Calculate the roots of the 2m-polynomial
        roots = np.roots(phi)
        # sort the roots according to their length (in descending order)
        roots_sorted = roots[np.argsort(abs(roots))[::-1]]

        z_0 = phi.sum() / np.poly1d(roots, True)(1)
        z_1_to_m = roots_sorted[:m]     # we need only those outside the unit␣
↪circle

        lambdas = 1 / z_1_to_m

        return z_1_to_m, z_0, lambdas


    def coeffs_of_c(self):
        '''
        This function computes the coefficients {c_j, j = 0, 1, ..., m} for
                c(z) = sum_{j = 0}^{m} c_j z^j

        Based on the expression (1.9). The order is
            c_coeffs = [c_0, c_1, ..., c_{m-1}, c_m]
        '''
        z_1_to_m, z_0 = self.roots_of_characteristic()[:2]
```

```
        c_0 = (z_0 * np.prod(z_1_to_m).real * (- 1)**self.m)**(.5)
        c_coeffs = np.poly1d(z_1_to_m, True).c * z_0 / c_0

        return c_coeffs[::-1]


    def solution(self):
        """
        This function calculates {lambda_j, j=1,...,m} and {A_j, j=1,...,m}
        of the expression (1.15)
        """
        lambdas = 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 - lambdas/lambdas[j]
            A[j] = c_0**(-2) / np.prod(denom[np.arange(self.m) != j])

        return lambdas, 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))
        phi_r = self.phi_r

        for i in range(N):
            for j in range(N):
                if abs(i-j) <= self.k:
                    V[i, j] = phi_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 is section␣
↪6 (1.59)
```

```
        It takes a realization for a^N, and the period in which the␣
↪prediciton is formed

        Output:  E[abar | a_t, a_{t-1}, ..., a_1, a_0]
        """

        N = np.asarray(a_hist).shape[0] - 1
        a_hist = np.asarray(a_hist).reshape(N + 1, 1)
        V = self.construct_V(N + 1)

        aux_matrix = np.zeros((N + 1, N + 1))
        aux_matrix[:(t + 1), :(t + 1)] = np.eye(t + 1)
        L = la.cholesky(V).T
        Ea_hist = la.inv(L) @ aux_matrix @ L @ a_hist

        return Ea_hist


    def optimal_y(self, a_hist, t = None):
        """
        - if t is NOT given it takes a_hist (list or numpy.array) as a␣
↪deterministic a_t
        - if t is given, it solves the combined control prediction problem␣
↪(section 7)
          (by default, t == None -> deterministic)

        for a given sequence of a_t (either 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 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 the lecutre
            Uy = np.linalg.solve(L, a_bar)            # U @ y_bar = L^{-1}a_
↪bar from the lecture
            y_bar = np.linalg.solve(U, Uy)            # y_bar = U^{-1}L^{-1}a_
↪bar

            # 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␣
↪lecutre
            Uy = np.linalg.solve(L, a_bar)            # U @ y_bar = L^{-1}a_
↪bar from the lecture
            y_bar = np.linalg.solve(U, Uy)            # y_bar = U^{-1}L^{-1}a_
↪bar

            # 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 ε_t is a serially uncorrelated random process with mean zero and variance unity

We want to use the Wiener-Kolmogorov formula (4.167) 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*

```
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

```
example.coeffs_of_c()
```

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

```
example.roots_of_characteristic()
```

```
(array([ 2.]), -2.0, array([ 0.5]))
```

Now let's form the covariance matrix of a time series vector of length N and put it in V

Then we'll take a Cholesky decomposition of $V = L^{-1}L^{-1} = LiLi'$ and use it to form the vector of "moving average representations" $x = Li\varepsilon$ and the vector of "autoregressive representations" $Lx = \varepsilon$

```
V = example.construct_V(N = 5)
print(V)
```

```
[[ 5. -2.  0.  0.  0.]
 [-2.  5. -2.  0.  0.]
 [ 0. -2.  5. -2.  0.]
 [ 0.  0. -2.  5. -2.]
 [ 0.  0.  0. -2.  5.]]
```

Notice how the lower rows of the "moving average representations" are converging to the appropriate infinite history Wold representation

```
Li = np.linalg.cholesky(V)
print(Li)
```

```
[[ 2.23606798  0.          0.          0.          0.        ]
 [-0.89442719  2.04939015  0.          0.          0.        ]
 [ 0.         -0.97590007  2.01186954  0.          0.        ]
 [ 0.          0.         -0.99410024  2.00293902  0.        ]
 [ 0.          0.          0.         -0.99853265  2.000733  ]]
```

Notice how the lower rows of the "autoregressive representations" are converging to the appropriate infinite history autoregressive representation

```
L = np.linalg.inv(Li)
print(L)
```

```
[[ 0.4472136   0.          0.          0.          0.        ]
 [ 0.19518001  0.48795004  0.          0.          0.        ]
 [ 0.09467621  0.23669053  0.49705012  0.          0.        ]
 [ 0.04698977  0.11747443  0.2466963   0.49926632  0.        ]
 [ 0.02345182  0.05862954  0.12312203  0.24917554  0.49981682]]
```

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

$$\theta(z) = \pi(z) \left(\frac{(z_1 z - 1)}{(z - z_1)} \right) \left(\frac{(z_2 z - 1)}{(z - z_2)} \right) \ldots \left(\frac{(z_k z - 1)}{(z - z_k)} \right)$$

The term multiplying $\pi(z)$ is termed a "Blaschke factor"

Then it can be proved directly that

$$\theta(z^{-1})\theta(z) = \pi(z^{-1})\pi(z)$$

and that the zeros of $\theta(z)$ are not inside the unit circle

Example 2

Consider a stochastic process X_t with moving average representation

$$X_t = (1 - \sqrt{2}L^2)\varepsilon_t$$

where ε_t is a serially uncorrelated random process with mean zero and variance unity

Let's find a Wold moving average representation for x_t

Let's use the Wiener-Kolomogorov formula (4.167) to compute the linear least squares forecasts $\hat{\mathbb{E}}\left[X_{t+j} \mid X_{t-1}, \ldots\right]$ for $j = 1, 2, 3$

We proceed in the same way as example 1

```
m = 2
y_m = np.asarray([.0, .0]).reshape(m, 1)
d = np.asarray([1, 0, -np.sqrt(2)])
r = np.asarray([1, 0, -np.sqrt(2)])
h = 0.0
example = LQFilter(d, h, y_m, r = d)
example.coeffs_of_c()
```

```
array([ 1.41421356, -0.        , -1.        ])
```

```
example.roots_of_characteristic()
```

```
(array([ 1.18920712, -1.18920712]),
 -1.4142135623731122,
 array([ 0.84089642, -0.84089642]))
```

```
V = example.construct_V(N = 8)
print(V)
```

```
[[ 3.          0.         -1.41421356  0.          0.          0.          0.␣
↪ 0.        ]
 [ 0.          3.          0.         -1.41421356  0.          0.          0.␣
↪ 0.        ]
 [-1.41421356  0.          3.          0.         -1.41421356  0.          0.␣
↪ 0.        ]
 [ 0.         -1.41421356  0.          3.          0.         -1.41421356  0.␣
↪ 0.        ]
 [ 0.          0.         -1.41421356  0.          3.          0.  -1.
↪41421356 0.        ]
 [ 0.          0.          0.         -1.41421356  0.          3.          0.␣
↪-1.41421356]
 [ 0.          0.          0.          0.         -1.41421356  0.          3.␣
↪ 0.        ]
 [ 0.          0.          0.          0.          0.         -1.41421356  0.␣
↪ 3.        ]]
```

```
Li = np.linalg.cholesky(V)
print(Li[-3:, :])
```

```
[[ 0.          0.          0.         -0.9258201   0.          1.46385011 0. ␣
↪        0.        ]
 [ 0.          0.          0.          0.         -0.96609178  0.         1.
↪43759058  0.        ]
 [ 0.          0.          0.          0.          0.         -0.96609178 0. ␣
↪        1.43759058]]
```

```
L = np.linalg.inv(Li)
print(L)
```

```
[[ 0.57735027  0.          0.          0.          0.          0.          0.␣
↪ 0.        ]
 [ 0.          0.57735027  0.          0.          0.          0.          0.␣
↪ 0.        ]
 [ 0.3086067   0.          0.65465367  0.          0.          0.          0.␣
↪ 0.        ]
 [ 0.          0.3086067   0.          0.65465367  0.          0.          0.␣
↪ 0.        ]
 [ 0.19518001  0.          0.41403934  0.          0.68313005  0.          0.␣
↪ 0.        ]
 [ 0.          0.19518001  0.          0.41403934  0.          0.68313005  0.␣
↪ 0.        ]
 [ 0.13116517  0.          0.27824334  0.          0.45907809  0.  0.69560834␣
↪ 0.        ]
 [ 0.          0.13116517  0.          0.27824334  0.          0.45907809  0.␣
↪ 0.69560834]]
```

Prediction

It immediately follows from the "orthogonality principle" of least squares (see *[AP91]* or *[Sar87]* [ch. X]) that

$$\begin{aligned}\hat{\mathbb{E}}[x_t \mid x_{t-m},\, x_{t-m+1}, \ldots x_1] &= \sum_{j=m}^{t-1} L_{t,t-j}^{-1}\, \varepsilon_{t-j} \\ &= [L_{t,1}^{-1}\, L_{t,2}^{-1},\, \ldots, L_{t,t-m}^{-1}\, 0\, 0 \ldots 0] L\, x\end{aligned} \tag{4.180}$$

This can be interpreted as a finite-dimensional version of the Wiener-Kolmogorov m-step ahead prediction formula

We can use (4.180) 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

$$\hat{\mathbb{E}}[x \mid x_s, x_{s-1}, \ldots, x_1] = L^{-1} \begin{bmatrix} I_s & 0 \\ 0 & 0_{(t-s)} \end{bmatrix} Lx \tag{4.181}$$

This formula will be convenient in representing the solution of control problems under uncertainty

Equation (4.177) can be recognized as a finite dimensional version of a moving average representation

Equation (4.176) can be viewed as a finite dimension version of an autoregressive representation

Notice that even if the x_t process is covariance stationary, so that V is such that V_{ij} depends only on $|i-j|$, the coefficients in the moving average representation are time-dependent, there being a different moving average for each t

If x_t is a covariance stationary process, the last row of L^{-1} converges to the coefficients in the Wold moving average representation for $\{x_t\}$ as $T \to \infty$

Further, if x_t is covariance stationary, for fixed k and $j > 0$, $L_{T,T-j}^{-1}$ converges to $L_{T-k,T-k-j}^{-1}$ as $T \to \infty$

That is, the "bottom" rows of L^{-1} converge to each other and to the Wold moving average coefficients as $T \to \infty$

This last observation gives one simple and widely-used practical way of forming a finite T approximation to a Wold moving average representation

First, form the covariance matrix $\mathbb{E}xx' = V$, then obtain the Cholesky decomposition $L^{-1}L^{-1'}$ of V, which can be accomplished quickly on a computer

The last row of L^{-1} gives the approximate Wold moving average coefficients

This method can readily be generalized to multivariate systems.

Combined Finite Dimensional Control and Prediction

Consider the finite-dimensional control problem, maximize

$$\mathbb{E} \sum_{t=0}^{N} \left\{ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} [d(L) y_t]^2 \right\}, \quad h > 0$$

where $d(L) = d_0 + d_1 L + \ldots + d_m L^m$, L is the lag operator, $\bar{a} = [a_N, a_{N-1} \ldots, a_1, a_0]'$ a random vector with mean zero and $\mathbb{E}\, \bar{a}\bar{a}' = V$

The variables $y_{-1}, \ldots, y_{-m}$ are given

Maximization is over choices of $y_0, y_1 \ldots, y_N$, where y_t is required to be a linear function of $\{y_{t-s-1}, t + m - 1 \geq 0;\ a_{t-s}, t \geq s \geq 0\}$

We saw in the lecture *Classical Control with Linear Algebra* that the solution of this problem under certainty could be represented in feedback-feedforward form

$$U\bar{y} = L^{-1}\bar{a} + K \begin{bmatrix} y_{-1} \\ \vdots \\ y_{-m} \end{bmatrix}$$

for some $(N+1) \times m$ matrix K

Using a version of formula (4.180), we can express $\hat{\mathbb{E}}[\bar{a} \mid a_s,\ a_{s-1}, \ldots, a_0]$ as

$$\hat{\mathbb{E}}[\bar{a} \mid a_s,\ a_{s-1}, \ldots, a_0] = \tilde{U}^{-1} \begin{bmatrix} 0 & 0 \\ 0 & I_{(s+1)} \end{bmatrix} \tilde{U}\bar{a}$$

where $I_{(s+1)}$ is the $(s+1) \times (s+1)$ identity matrix, and $V = \tilde{U}^{-1}\tilde{U}^{-1'}$, where $\tilde{U}$ is the *upper* trangular 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 (4.181) by replacing L with L^T

The optimal decision rule to use at time $0 \leq t \leq N$ is then given by the $(N - t + 1)^{\text{th}}$ row of

$$U\bar{y} = L^{-1}\tilde{U}^{-1} \begin{bmatrix} 0 & 0 \\ 0 & I_{(t+1)} \end{bmatrix} \tilde{U}\bar{a} + K \begin{bmatrix} y_{-1} \\ \vdots \\ y_{-m} \end{bmatrix}$$

Exercises

Exercise 1

Let $Y_t = (1 - 2L)u_t$ where u_t is a mean zero white noise with $\mathbb{E}u_t^2 = 1$. Let

$$X_t = Y_t + \varepsilon_t$$

where ε_t is a serially uncorrelated white noise with $\mathbb{E}\varepsilon_t^2 = 9$, and $\mathbb{E}\varepsilon_t u_s = 0$ for all t and s

Find the Wold moving average representation for X_t

Find a formula for the A_{1j}'s in

$$\mathbb{E}\widehat{X}_{t+1} \mid X_t, X_{t-1}, \ldots = \sum_{j=0}^{\infty} A_{1j} X_{t-j}$$

Find a formula for the A_{2j}'s in

$$\hat{\mathbb{E}} X_{t+2} \mid X_t, X_{t-1}, \ldots = \sum_{j=0}^{\infty} A_{2j} X_{t-j}$$

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 ε_t be an $n \times 1$ vector white noise with mean 0 and contemporaneous covariance matrix H, where H is a positive definite matrix

Let $X_t = Y_t + \varepsilon_t$

Define the covariograms as $C_X(\tau) = \mathbb{E}X_t X_{t-\tau}', C_Y(\tau) = \mathbb{E}Y_t Y_{t-\tau}', C_{YX}(\tau) = \mathbb{E}Y_t X_{t-\tau}'$

Then define the matrix covariance generating function, as in (4.150), only interpret all the objects in (4.150) as matrices

Show that the covariance generating functions are given by

$$\begin{aligned} g_y(z) &= D(z)D(z^{-1})' \\ g_X(z) &= D(z)D(z^{-1})' + H \\ g_{YX}(z) &= D(z)D(z^{-1})' \end{aligned}$$

A factorization of $g_X(z)$ can be found (see *[Roz67]* or *[Whi83]*) of the form

$$D(z)D(z^{-1})' + H = C(z)C(z^{-1})', \quad C(z) = \sum_{j=0}^{m} C_j z^j$$

where the zeros of $|C(z)|$ do not lie inside the unit circle

A vector Wold moving average representation of X_t is then

$$X_t = C(L)\eta_t$$

where η_t is an $(n \times 1)$ vector white noise that is "fundamental" for X_t

That is, $X_t - \hat{\mathbb{E}}\left[X_t \mid X_{t-1}, X_{t-2} \ldots\right] = C_0\, \eta_t$

The optimum predictor of X_{t+j} is

$$\hat{\mathbb{E}}\left[X_{t+j} \mid X_t, X_{t-1}, \ldots\right] = \left[\frac{C(L)}{L^j}\right]_+ \eta_t$$

If $C(L)$ is invertible, i.e., if the zeros of $\det C(z)$ lie strictly outside the unit circle, then this formula can be written

Discrete State Dynamic Programming

Contents

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

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 libary

How to Read this Lecture

We have used dynamic programming in a number of earlier lectures, such as

- The *shortest path lecture*
- The *McCall growth model lecture*
- The *optimal growth lecture*

Here we shift to a more systematic and theoretical treatment, including algorithms and implementation

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,

- *[LS12]*

- *[HLL96]*, section 3.5
- *[Put05]*
- *[SLP89]*
- *[Rus96]*
- *[MF02]*
- EDTC, chapter 5

Discrete DPs

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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t r(s_t, a_t) \tag{4.182}$$

where

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

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

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

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

Examples:

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

Policies

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

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

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

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

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

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

- See section 5.5 of *[Put05]* for discussion and proofs

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

The aim is to find an optimal policy, in the sense of one that maximizes (4.182)

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

$$SA := \{(s, a) \mid s \in S,\ a \in A(s)\} of * feasible state - action pairs *$$

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

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

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

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

Denote the set of all feasible policies by Σ

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

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

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

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

Notice that Q_σ is a *stochastic matrix* on S

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

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

$$(Q_\sigma^t r_\sigma)(s) = \mathbb{E}[r(s_t, \sigma(s_t)) \mid s_0 = s] \quad \text{when } \{s_t\} \sim Q_\sigma \tag{4.183}$$

Comments

- $\{s_t\} \sim Q_\sigma$ means that the state is generated by stochastic matrix Q_σ

- See *this discussion* on computing expectations of Markov chains for an explanation of the expression in (4.183)

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

This is natural because they are in one to one correspondence

Value and Optimality

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

To calculate this quantity we pass the expectation through the sum in (4.182) and use (4.183) to get

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

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

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

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

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

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

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

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

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

Two Operators

It is useful to define the following operators:

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

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

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

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

This can be written more succinctly in operator notation as

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

The two operators are both monotone

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

They are also contraction mappings with modulus β

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

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

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

The Bellman Equation and the Principle of Optimality

The main principle of the theory of dynamic programming is that

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

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

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

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

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

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

Solving Discrete DPs

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

Code for solving dicrete 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 $v^0 \colon S \to \mathbb{R}$ converges to v^*

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

Policy Function Iteration

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

Each iteration consists of

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

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

- See theorem 10.2.6 of EDTC for a proof

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

Modified Policy Function Iteration

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

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

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

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

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

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

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

$$Q(s, a, s') := \begin{cases} \frac{1}{B+1} & \text{if } a \leq s' \leq a + B \\ 0 & \text{otherwise} \end{cases} \tag{4.184}$$

Defining a DiscreteDP Instance

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

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

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

For Q we follow the rule in (4.184)

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

A simple class that sets up these objects for us in the current application `can be found in`

For convenience let's repeat it here:

```
"""
A simple optimal growth model, for testing the DiscreteDP class.

Filename: finite_dp_og_example.py
"""
import numpy as np

class SimpleOG:
```

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

        self.B, self.M, self.alpha, self.beta  = B, M, alpha, beta
        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.alpha

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

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

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

        and zero otherwise.
        """

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

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

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

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

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

```
import quantecon as qe

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

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

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

Let's see what we've got here

```
dir(results)
```

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

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

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

```
results.v
```

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

```
results.sigma
```

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

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

Let's make sure this didn't happen

```
results.max_iter
```

```
250
```

```
results.num_iter
```

```
3
```

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

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

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

```
results.mc.stationary_distributions
```

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

Here's the same information in a bar graph

What happens if the agent is more patient?

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

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

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

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 exercise solution notebook below)

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

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

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

```
import quantecon as qe
import numpy as np

B, M, alpha, beta = 10, 5, 0.5, 0.9
n = B + M + 1
m = M + 1
def u(c):
    return c**alpha

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, beta, s_indices, a_indices)
```

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

Exercises

In the deterministic optimal growth *dynamic programming lecture*, we solved a *benchmark model* that has an analytical solution to check we could replicate it numerically

The exercise is to replicate this solution using `DiscreteDP`

Solutions

Written jointly with Diasuke Oyama

Let's start with some imports

```
import numpy as np
import scipy.sparse as sparse
import matplotlib.pyplot as plt
from quantecon import compute_fixed_point
from quantecon.markov import DiscreteDP
```

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 $\beta = 0.95$

```
alpha = 0.65
f = lambda k: k**alpha
u = np.log
beta = 0.95
```

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

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

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

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

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

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

Thus the Bellman equation is:

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

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

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

We first construct indices for state-action pairs:

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

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

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

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

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

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

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

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

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

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

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

Discrete growth model:

```
ddp = DiscreteDP(R, Q, 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:

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

```
10
```

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

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

# Exact solution of the continuous version
ab = alpha * beta
c1 = (np.log(1 - ab) + np.log(ab) * ab / (1 - ab)) / (1 - beta)
c2 = alpha / (1 - ab)
def v_star(k):
    return c1 + c2 * np.log(k)

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

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

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

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

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

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

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

The outcomes appear very close to those of the continuous version

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

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

```
121.49819147053378
```

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

```
0.012681735127500815
```

The optimal consumption functions are close as well:

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

```
0.0038265231000100819
```

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

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

```
False
```

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

```
174
```

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

```
0.0019618533397668392
```

The value function is monotone:

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

```
True
```

Comparison of the solution methods

Let us solve the problem by the other two methods

Value iteration

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

```
294
```

```
np.array_equal(sigma, res1.sigma)
```

```
True
```

Modified policy iteration

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

```
16
```

```
np.array_equal(sigma, res2.sigma)
```

```
True
```

Speed comparison

```
%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`

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

plt.show()
```

We next plot the consumption policies along the value iteration.

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

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

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

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

    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('{} value function iterations'.format(n))
plt.show()
```

```
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
```

```
1            5.518e+00      1.464e-03
2            4.070e+00      2.330e-03
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
1            5.518e+00      1.046e-03
2            4.070e+00      1.970e-03
3            3.866e+00      2.848e-03
4            3.673e+00      3.715e-03
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
1            5.518e+00      1.052e-03
2            4.070e+00      1.925e-03
3            3.866e+00      2.804e-03
4            3.673e+00      3.654e-03
5            3.489e+00      4.510e-03
6            3.315e+00      5.350e-03
```

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

Dynamics of the capital stock

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

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

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

sample_size = 25

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

# Create a new instance, not to modify the one used above
ddp0 = DiscreteDP(R, Q, beta, 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=r'$\beta = {}$'.
↪format(beta))

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

Appendix: Algorithms

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

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

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

Value Iteration

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

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

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

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

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

Policy Iteration

The `DiscreteDP` policy iteration method runs as follows

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

The policy iteration algorithm terminates in a finite number of iterations

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

Modified Policy Iteration

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

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

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

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

See also the documentation for `DiscreteDP`

CHAPTER

FIVE

PART 5: SINGLE AGENT MODELS

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.

Shortest Paths

Contents

- *Shortest Paths*
  - *Overview*
  - *Outline of the Problem*
  - *Finding Least-Cost Paths*
  - *Solving for J*
  - *Exercises*
  - *Solutions*

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

Outline of the Problem

The shortest path problem is one of finding how to traverse a graph from one specified node to another at minimum cost

Consider the following graph

We wish to travel from node (vertex) A to node G at minimum cost

- Arrows (edges) indicate the movements we can take
- Numbers next to 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
- A, D, F, G at cost 8

Finding Least-Cost Paths

For large graphs we need a systematic solution

Let $J(v)$ denote the minimum cost-to-go from node v, understood as the total cost from v if we take the best route

Suppose that we know $J(v)$ for each node v, as shown below for the graph from the preceding example

Note that $J(G) = 0$

Intuitively, the best path can now be found as follows

- Start at A
- From node v, move to any node that solves

$$\min_{w \in F_v} \{c(v, w) + J(w)\} \tag{5.1}$$

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

$$J(v) = \min_{w \in F_v} \{c(v, w) + J(w)\} \tag{5.2}$$

This is known as the Bellman equation

- That is, J is the solution to the Bellman equation
- There are algorithms for computing the minimum cost-to-go function J

Solving for J

The standard algorithm for finding J is to start with

$$J_0(v) = M \text{ if } v \neq \text{ destination, else } J_0(v) = 0 \tag{5.3}$$

where M is some large number

Now we use the following algorithm

1. Set $n = 0$
2. Set $J_{n+1}(v) = \min_{w \in F_v} \{c(v, w) + J_n(w)\}$ for all v
3. If J_{n+1} and J_n are not equal then increment n, go to 2

In general, this sequence converges to J—the proof is omitted

Exercises

Exercise 1

Use the algorithm given above to find the optimal path (and its cost) for this graph

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

Solutions

Exercise 1

Here's the data

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*

```
%%file graph.txt
node0, node1 0.04, node8 11.11, node14 72.21
node1, node46 1247.25, node6 20.59, node13 64.94
node2, node66 54.18, node31 166.80, node45 1561.45
node3, node20 133.65, node6 2.06, node11 42.43
node4, node75 3706.67, node5 0.73, node7 1.02
node5, node45 1382.97, node7 3.33, node11 34.54
node6, node31 63.17, node9 0.72, node10 13.10
node7, node50 478.14, node9 3.15, node10 5.85
node8, node69 577.91, node11 7.45, node12 3.18
node9, node70 2454.28, node13 4.42, node20 16.53
node10, node89 5352.79, node12 1.87, node16 25.16
node11, node94 4961.32, node18 37.55, node20 65.08
node12, node84 3914.62, node24 34.32, node28 170.04
node13, node60 2135.95, node38 236.33, node40 475.33
node14, node67 1878.96, node16 2.70, node24 38.65
node15, node91 3597.11, node17 1.01, node18 2.57
node16, node36 392.92, node19 3.49, node38 278.71
node17, node76 783.29, node22 24.78, node23 26.45
node18, node91 3363.17, node23 16.23, node28 55.84
node19, node26 20.09, node20 0.24, node28 70.54
node20, node98 3523.33, node24 9.81, node33 145.80
node21, node56 626.04, node28 36.65, node31 27.06
node22, node72 1447.22, node39 136.32, node40 124.22
node23, node52 336.73, node26 2.66, node33 22.37
node24, node66 875.19, node26 1.80, node28 14.25
node25, node70 1343.63, node32 36.58, node35 45.55
node26, node47 135.78, node27 0.01, node42 122.00
node27, node65 480.55, node35 48.10, node43 246.24
node28, node82 2538.18, node34 21.79, node36 15.52
node29, node64 635.52, node32 4.22, node33 12.61
node30, node98 2616.03, node33 5.61, node35 13.95
node31, node98 3350.98, node36 20.44, node44 125.88
```

```
node32, node97 2613.92, node34 3.33, node35 1.46
node33, node81 1854.73, node41 3.23, node47 111.54
node34, node73 1075.38, node42 51.52, node48 129.45
node35, node52 17.57, node41 2.09, node50 78.81
node36, node71 1171.60, node54 101.08, node57 260.46
node37, node75 269.97, node38 0.36, node46 80.49
node38, node93 2767.85, node40 1.79, node42 8.78
node39, node50 39.88, node40 0.95, node41 1.34
node40, node75 548.68, node47 28.57, node54 53.46
node41, node53 18.23, node46 0.28, node54 162.24
node42, node59 141.86, node47 10.08, node72 437.49
node43, node98 2984.83, node54 95.06, node60 116.23
node44, node91 807.39, node46 1.56, node47 2.14
node45, node58 79.93, node47 3.68, node49 15.51
node46, node52 22.68, node57 27.50, node67 65.48
node47, node50 2.82, node56 49.31, node61 172.64
node48, node99 2564.12, node59 34.52, node60 66.44
node49, node78 53.79, node50 0.51, node56 10.89
node50, node85 251.76, node53 1.38, node55 20.10
node51, node98 2110.67, node59 23.67, node60 73.79
node52, node94 1471.80, node64 102.41, node66 123.03
node53, node72 22.85, node56 4.33, node67 88.35
node54, node88 967.59, node59 24.30, node73 238.61
node55, node84 86.09, node57 2.13, node64 60.80
node56, node76 197.03, node57 0.02, node61 11.06
node57, node86 701.09, node58 0.46, node60 7.01
node58, node83 556.70, node64 29.85, node65 34.32
node59, node90 820.66, node60 0.72, node71 0.67
node60, node76 48.03, node65 4.76, node67 1.63
node61, node98 1057.59, node63 0.95, node64 4.88
node62, node91 132.23, node64 2.94, node76 38.43
node63, node66 4.43, node72 70.08, node75 56.34
node64, node80 47.73, node65 0.30, node76 11.98
node65, node94 594.93, node66 0.64, node73 33.23
node66, node98 395.63, node68 2.66, node73 37.53
node67, node82 153.53, node68 0.09, node70 0.98
node68, node94 232.10, node70 3.35, node71 1.66
node69, node99 247.80, node70 0.06, node73 8.99
node70, node76 27.18, node72 1.50, node73 8.37
node71, node89 104.50, node74 8.86, node91 284.64
node72, node76 15.32, node84 102.77, node92 133.06
node73, node83 52.22, node76 1.40, node90 243.00
node74, node81 1.07, node76 0.52, node78 8.08
node75, node92 68.53, node76 0.81, node77 1.19
node76, node85 13.18, node77 0.45, node78 2.36
node77, node80 8.94, node78 0.98, node86 64.32
node78, node98 355.90, node81 2.59
node79, node81 0.09, node85 1.45, node91 22.35
node80, node92 121.87, node88 28.78, node98 264.34
node81, node94 99.78, node89 39.52, node92 99.89
node82, node91 47.44, node88 28.05, node93 11.99
node83, node94 114.95, node86 8.75, node88 5.78
node84, node89 19.14, node94 30.41, node98 121.05
```

```
node85, node97 94.51, node87 2.66, node89 4.90
node86, node97 85.09
node87, node88 0.21, node91 11.14, node92 21.23
node88, node93 1.31, node91 6.83, node98 6.12
node89, node97 36.97, node99 82.12
node90, node96 23.53, node94 10.47, node99 50.99
node91, node97 22.17
node92, node96 10.83, node97 11.24, node99 34.68
node93, node94 0.19, node97 6.71, node99 32.77
node94, node98 5.91, node96 2.03
node95, node98 6.17, node99 0.27
node96, node98 3.32, node97 0.43, node99 5.87
node97, node98 0.30
node98, node99 0.33
node99,
```

```
Writing graph.txt
```

Here's our solution

```
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.
    """
    graph = {}
    infile = open(in_file)
    for line in infile:
        elements = line.split(',')
        node = elements.pop(0).strip()
        graph[node] = []
        if node != 'node99':
            for element in elements:
                destination, cost = element.split()
                graph[node].append((destination.strip(), float(cost)))
    infile.close()
    return graph

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:
    J[node] = M
J['node99'] = 0

while 1:
    next_J = update_J(J, graph)
    if next_J == J:
        break
    else:
        J = next_J
print_best_path(J, graph)
```

```
node0
node8
node11
node18
node23
node33
node41
node53
node56
node57
node60
node67
node70
node73
node76
```

```
node85
node87
node88
node93
node94
node96
node97
node98
node99

Cost:  160.55000000000007
```

The McCall Job Search Model

Contents

- *The McCall Job Search Model*
  - *Overview*
  - *The Model*
  - *Solving the Model using Dynamic Programming*
  - *Implementation*
  - *The Reservation Wage*
  - *Exercises*
  - *Solutions*

Overview

The McCall search model *[McC70]* helped transform economists' way of thinking about labor markets

It did this by casting

- the loss of a job as a capital loss, and
- a spell of unemployment as an *investment* in searching for an acceptable job

To solve the model, we follow McCall in using dynamic programming

Dynamic programming was discussed previously in the *lecture on shortest paths*

The McCall model is a nice vehicle for readers to start to make themselves more comfortable with this approach to optimization

(More extensive and more formal treatments of dynamic programming are given in later lectures)

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

It is assumed that the worker lives forever

He can be in one of two states: employed or unemployed

He wants to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(y_t) \tag{5.4}$$

which represents the expected value of the discounted utility of his income

The constant β lies in $(0, 1)$ and is called a **discount factor**

The smaller is β, the more the worker discounts future utility relative to current utility

The variable y_t is

- his wage w_t when employed
- unemployment compensation c when unemployed

The function u is a utility function satisfying $u' > 0$ and $u'' < 0$

Timing and Decisions

Let's consider what happens at the start of a given period (e.g., a month, if the timing of the model is monthly)

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

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

(Note that we do not allow for job search while employed—this topic is taken up in a *later lecture*)

Job Termination

When employed, he faces a constant probability α of becoming unemployed at the end of the period

Solving the Model using Dynamic Programming

As promised, we shall solve the McCall search model using dynamic programming

Dynamic programming is an ingenious method for solving a problem that starts by

1. assuming that you know the answer,
2. writing down some natural conditions that the answer must satisfy, then
3. solving those conditions to find the answer

So here goes

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 (5.4) when the worker makes optimal decisions now and 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

In this case, a little thought will convince you that V and U should satisfy

$$V(w) = u(w) + \beta[(1-\alpha)V(w) + \alpha U] \tag{5.5}$$

and

$$U = u(c) + \beta \sum_{w'} \max\left\{U, V(w')\right\} p(w') \tag{5.6}$$

The sum is over all possible wage values, which we assume for convenience is finite

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 (5.5) and (5.6) are the values of a worker in a particular situation *today*
- The right hand sides of the equations are the discounted (by β) 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 (5.6) 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 (5.5) and (5.6) are called *Bellman equations* after the mathematician Richard Bellman

It turns out that equations (5.5) and (5.6) provide enough information to solve out for both V and U

Before discussing this, however, let's make a small extension to the model

Stochastic Offers

Let's suppose now that unemployed workers don't always receive job offers

Instead, let's suppose that unemployed workers only receive an offer with probability γ

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

$$V(w) = u(w) + \beta[(1-\alpha)V(w) + \alpha U] \tag{5.7}$$

and

$$U = u(c) + \beta(1-\gamma)U + \beta\gamma \sum_{w'} \max\left\{U, V(w')\right\} p(w') \tag{5.8}$$

Solving the Bellman Equations

The Bellman equations are nonlinear in U and V, and hence not trivial to solve

One way to solve them is to

1. make guesses for U and V
2. plug these guesses into the right hand sides of (5.7) and (5.8)
3. update the left hand sides from this rule and then repeat

In other words, we are iterating using the rules

$$V_{n+1}(w) = u(w) + \beta[(1-\alpha)V_n(w) + \alpha U_n] \tag{5.9}$$

and

$$U_{n+1} = u(c) + \beta(1-\gamma)U_n + \beta\gamma \sum_{w'} \max\{U_n, V_n(w')\}p(w') \tag{5.10}$$

starting from some initial conditions U_0, V_0

This procedure is called *iterating on the Bellman equations*

It turns out that these iterations are guaranteed to converge to the V and U that solve (5.7) and (5.8)

We discuss the theory behind this property extensively in later lectures (see, e.g., the discussion in *this lecture*)

For now let's try implementing the iteration scheme to see what the solutions look like

Implementation

Code to iterate on the Bellman equations can be found in mccall_bellman_iteration.py

We repeat it here for convenience

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

You'll see that in places we use *just in time compilation via Numba* to achieve good performance

```
"""
Implements iteration on the Bellman equations to solve the McCall growth model

"""

import numpy as np
from quantecon.distributions import BetaBinomial
from numba import jit

# A default utility function

@jit
def u(c, sigma):
    if c > 0:
        return (c**(1 - sigma) - 1) / (1 - sigma)
    else:
        return -10e6


class McCallModel:
    """
    Stores the parameters and functions associated with a given model.
    """

    def __init__(self, alpha=0.2,     # Job separation rate
                       beta=0.98,     # Discount rate
                       gamma=0.7,     # Job offer rate
                       c=6.0,         # Unemployment compensation
                       sigma=2.0,     # Utility parameter
                       w_vec=None,    # Possible wage values
                       p_vec=None): # Probabilities over w_vec

        self.alpha, self.beta, self.gamma, self.c = alpha, beta, gamma, c
        self.sigma = sigma

        # 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(alpha, beta, gamma, c, sigma, 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, sigma) + beta * ((1 - alpha) * V[w_idx] + alpha *␣
↪U)

    U_new = u(c, sigma) + beta * (1 - gamma) * U + \
                    beta * gamma * np.sum(np.maximum(U, V) * p_vec)

    return U_new


def solve_mccall_model(mcm, tol=1e-5, max_iter=2000):
    """
    Iterates to convergence on the Bellman equations

    Parameters
    ----------
    mcm : an instance of McCallModel
    tol : float
        error tolerance
    max_iter : int
        the maximum number of iterations
    """

    V = np.ones(len(mcm.w_vec))  # Initial guess of V
    V_new = np.empty_like(V)     # To store updates to V
    U = 1                        # Initial guess of U
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        U_new = _update_bellman(mcm.alpha, mcm.beta, mcm.gamma,
                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
```

The approch 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

```
import matplotlib.pyplot as plt

mcm = McCallModel()
V, U = solve_mccall_model(mcm)

fig, ax = plt.subplots()

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

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
```

Let's use it to look at how the reservation wage varies with parameters

In each instance below we'll show you a figure and then ask you to reproduce it in the exercises

The Reservation Wage and Unemployment Compensation

First, let's look at how $\bar{w}$ varies with unemployment compensation

In the figure below, we use the default parameters in the *McCallModel* class, apart from c (which takes the values given on the horizonal 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 β

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 $\bar{w}$ varies with the job separation rate α

Higher α translates to a greater chance that a worker will face termination in each period once employed

Once more, the results are in line with our intuition

If the separation rate is high, then the benefit of holding out for a higher wage falls

Hence the reservation wage is lower

Exercises

Exercise 1

Reproduce all the reservation wage figures shown above

Exercise 2

Plot the reservation wage against the job offer rate γ

Use

```
grid_size = 25
gamma_vals = np.linspace(0.05, 0.95, grid_size)
```

Interpret your results

Solutions

Exercise 1

Using the *compute_reservation_wage* function mentioned earlier in the lecture, we can create an array for reserveration wages for different values of c, β and α and plot the results like so

```
"""
Plots reservation wage against unemployment compensation

"""

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()

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 γ as follows

```
"""
Plots reservation wage against gamma

"""
import matplotlib.pyplot as plt

grid_size = 25
gamma_vals = np.linspace(0.05, 0.95, grid_size)
w_bar_vals = np.empty_like(gamma_vals)

mcm = McCallModel()

fig, ax = plt.subplots(figsize=(9, 6))

for i, gamma in enumerate(gamma_vals):
    mcm.gamma = gamma
```

```
    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(gamma_vals.min(), gamma_vals.max())
txt = r'$\bar w$ as a function of $\gamma$'
ax.plot(gamma_vals, w_bar_vals, 'b-', lw=2, alpha=0.7, label=txt)
ax.legend(loc='upper left')
ax.grid()

plt.show()
```

As expected, the reservation wage increases in γ

This is because higher γ translates to a more favorable job search environment

Hence workers are less willing to accept lower offers

A Stochastic Optimal Growth Model

Contents

- *A Stochastic Optimal Growth Model*
  - *Overview*
  - *The Model*
  - *Computation*
  - *Exercises*
  - *Solutions*

Overview

In this lecture we're going to study a simple optimal growth model with one agent

The model is a version of the standard one sector infinite horizon growth model studied in

- *[SLP89]*, chapter 2
- *[LS12]*, 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'll discuss some of the technical details of dynamic programming as we go along

The Model

Consider an agent who owns an amount $y_t \in \mathbb{R}_+ := [0, \infty)$ of a consumption good at time t

This output can either be consumed or invested

When the good is invested it is transformed one-for-one into capital

The resulting capital stock, denoted here by k_{t+1}, will then be used for production

Production is stochastic, in that it also depends on a shock ξ_{t+1} realized at the end of the current period

Next period output is

$$y_{t+1} := f(k_{t+1})\xi_{t+1}$$

where $f \colon \mathbb{R}_+ \to \mathbb{R}_+$ is called the production function

The resource constraint is

$$k_{t+1} + c_t \leq y_t \tag{5.11}$$

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 ξ_t will be denoted ϕ
- The production function f is assumed to be increasing and continuous
- Depreciation of capital is not made explicit but can be incorporated into the production function

While many other treatments of the stochastic growth model use k_t as the state variable, we will use y_t

This will allow us to treat a stochastic model while maintaining only one state variable

We consider alternative states and timing specifications in some of our other lectures

Optimization

Taking y_0 as given, the agent wishes to maximize

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(c_t)\right] \tag{5.12}$$

subject to

$$y_{t+1} = f(y_t - c_t)\xi_{t+1} \quad \text{and} \quad 0 \leq c_t \leq y_t \quad \text{for all } t \tag{5.13}$$

where

- u is a bounded, continuous and strictly increasing utility function and
- $\beta \in (0, 1)$ is a discount factor

In (5.13) we are assuming that the resource constraint (5.11) holds with equality — which is reasonable because u is strictly increasing and no output will be wasted at the optimum

In summary, the agent's aim is to select a path $c_0, c_1, c_2, \ldots$ for consumption that is

1. nonnegative,
2. feasible in the sense of (5.11),
3. optimal, in the sense that it maximizes (5.12) 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 ξ_{t+1}

In the present context

- y_t is called the *state* variable — it summarizes the "state of the world" at the start of each period
- c_t is called the *control* variable — a value chosen by the agent each period after observing the state

The Policy Function Approach

One way to think about solving this problem is to look for the best **policy function**

A policy function is a map from past and present observables into current action

We'll be particularly interested in **Markov policies**, which are maps from the current state y_t into a current action c_t

For dynamic programming problems such as this one (in fact for any Markov decision process), the optimal policy is always a Markov policy

In other words, the current state y_t provides a sufficient statistic for the history in terms of making an optimal decision today

This is quite intuitive but if you wish you can find proofs in texts such as *[SLP89]* (section 4.1)

Hereafter we focus on finding the best Markov policy

In our context, a Markov policy is a function $\sigma \colon \mathbb{R}_+ \to \mathbb{R}_+$, with the understanding that states are mapped to actions via

$$c_t = \sigma(y_t) \quad \text{for all } t$$

In what follows, we will call σ a *feasible consumption policy* if it satisfies

$$0 \leq \sigma(y) \leq y \quad \text{for all} \quad y \in \mathbb{R}_+ \tag{5.14}$$

In other words, a feasible consumption policy is a Markov policy that respects the resource constraint

The set of all feasible consumption policies will be denoted by Σ

Each $\sigma \in \Sigma$ determines a *continuous state Markov process* $\{y_t\}$ for output via

$$y_{t+1} = f(y_t - \sigma(y_t))\xi_{t+1}, \quad y_0 \text{ given} \tag{5.15}$$

This is the time path for output when we choose and stick with the policy σ

We insert this process into the objective function to get

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(c_t)\right] = \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(\sigma(y_t))\right] \tag{5.16}$$

This is the total expected present value of following policy σ forever, given initial income y_0

The aim is to select a policy that makes this number as large as possible

The next section covers these ideas more formally

Optimality

The **policy value function** v_σ associated with a given policy σ is the mapping defined by

$$v_\sigma(y) = \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(\sigma(y_t))\right] \tag{5.17}$$

when $\{y_t\}$ is given by (5.15) with $y_0 = y$

In other words, it is the lifetime value of following policy σ starting at initial condition y

The **value function** is then defined as

$$v^*(y) := \sup_{\sigma \in \Sigma} \; v_\sigma(y) \tag{5.18}$$

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 (5.18) for all $y \in \mathbb{R}_+$

The Bellman Equation

With our assumptions on utility and production function, the value function as defined in (5.18) also satisfies a **Bellman equation**

For this problem, the Bellman equation takes the form

$$w(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y-c)z)\phi(dz) \right\} \qquad (y \in \mathbb{R}_+) \tag{5.19}$$

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, (5.19) 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

$$\max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y-c)z)\phi(dz) \right\} \tag{5.20}$$

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 (5.19)

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

$$Tw(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y-c)z)\phi(dz) \right\} \qquad (y \in \mathbb{R}_+) \tag{5.21}$$

In other words, T sends the function w into the new function Tw defined (5.21)

By construction, the set of solutions to the Bellman equation (5.19) *exactly coincides with* the set of fixed points of T

For example, if $Tw = w$, then, for any $y \geq 0$,

$$w(y) = Tw(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y-c)z)\phi(dz) \right\}$$

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

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]*

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 (5.21), 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 (5.21)

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 (5.21)
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 $0, 0.2, 0.4, 0.6, 0.8, 1$

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

```
import numpy as np
from scipy.optimize import fminbound


def bellman_operator(w, grid, beta, 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
    beta : 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:
        sigma = np.empty_like(w)

    # == set Tw[i] = max_c { u(c) + beta E w(f(y  - c) z)} == #
    for i, y in enumerate(grid):
        def objective(c):
            return - u(c) - beta * np.mean(w_func(f(y - c) * shocks))
        c_star = fminbound(objective, 1e-10, y)
        if compute_policy:
            sigma[i] = c_star
        Tw[i] = - objective(c_star)

    if compute_policy:
        return Tw, sigma
    else:
        return Tw
```

The arguments to *bellman_operator* are described in the docstring to the function

Notice that the expectation in (5.21) 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 ϕ

Monte Carlo is not always the most efficient way to compute integrals numerically but it does have some theoretical advantages in the present setting

(For example, it preserves the contraction mapping property of the Bellman operator — see, e.g., *[PalS13]*)

An Example

Let's test out our operator when

- $f(k) = k^\alpha$
- $u(c) = \ln c$
- ϕ is the distribution of $\exp(\mu + \sigma\zeta)$ when ζ is standard normal

As is well-known (see *[LS12]*, section 3.1.2), for this particular problem an exact analytical solution is available, with

$$v^*(y) = \frac{\ln(1-\alpha\beta)}{1-\beta} + \frac{(\mu + \alpha\ln(\alpha\beta))}{1-\alpha}\left[\frac{1}{1-\beta} - \frac{1}{1-\alpha\beta}\right] + \frac{1}{1-\alpha\beta}\ln y \tag{5.22}$$

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 ~ LN(mu, s).

    The class holds parameters and true value and policy functions.
    """

    def __init__(self, alpha=0.4, beta=0.96, mu=0, s=0.1):

        self.alpha, self.beta, self.mu, self.s = alpha, beta, mu, s

        # == Some useful constants == #
        self.ab = alpha * beta
        self.c1 = np.log(1 - self.ab) / (1 - beta)
        self.c2 = (mu + alpha * np.log(self.ab)) / (1 - alpha)
        self.c3 = 1 / (1 - beta)
        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.alpha
```

```
    def f_prime(self, k):
        return self.alpha * k**(self.alpha - 1)

    def c_star(self, y):
        " True optimal policy.  "
        return (1 - self.alpha * self.beta) * 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

```
lg = LogLinearOG()
# == Unpack parameters / functions for convenience == #
alpha, beta, mu, s = lg.alpha, lg.beta, lg.mu, lg.s
v_star = lg.v_star
```

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(mu + s * np.random.randn(shock_size))
```

Now let's do some tests

In the code shown below and all other code, we assume that you've already run the code above, as well as

```
import matplotlib.pyplot as plt
```

As one preliminary test, let's 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,
                     beta,
                     np.log,
                     lambda k: k**alpha,
```

```
                        shocks)

fig, ax = plt.subplots(figsize=(9, 5))
ax.set_ylim(-35, -24)
ax.plot(grid, w, lw=2, alpha=0.6, label=r'$Tv^*$')
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'll start with is $w(y) = 5\ln(y)$

```
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

Let's try a more serious attempt to converge to v^*, using QuantEcon's *compute_fixed_point* function

```
from quantecon import compute_fixed_point

Tw = np.empty(len(grid))
initial_w = 5 * np.log(grid)

# Turn the Bellman operator into a function of one variable
```

```
T = lambda w: bellman_operator(w,
                               grid,
                               beta,
                               np.log,
                               lambda k: k**alpha,
                               shocks,
                               compute_policy=False)

v_star_approx = compute_fixed_point(T, initial_w,
                                    error_tol=1e-5,  # error_tol
                                    max_iter=500,   # max_iter
                                    verbose=2,  # verbose
                                    print_skip=10,     # print_skip
                                    method='iteration')
```

Here's the output

```
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
5            6.666e+00      4.999e-01
10           4.747e+00      9.615e-01
15           3.396e+00      1.423e+00
20           2.435e+00      1.846e+00
25           1.750e+00      2.275e+00
30           1.260e+00      2.924e+00
35           9.087e-01      3.431e+00
40           6.570e-01      3.861e+00
45           4.761e-01      4.288e+00
50           3.459e-01      4.720e+00
55           2.519e-01      5.146e+00
60           1.839e-01      5.578e+00
65           1.346e-01      6.004e+00
70           9.872e-02      6.473e+00
75           7.262e-02      6.953e+00
80           5.355e-02      7.435e+00
85           3.959e-02      7.863e+00
90           2.934e-02      8.290e+00
95           2.180e-02      8.863e+00
100          1.624e-02      9.295e+00
105          1.212e-02      9.730e+00
110          9.073e-03      1.017e+01
115          6.804e-03      1.063e+01
120          5.114e-03      1.106e+01
125          3.851e-03      1.149e+01
130          2.906e-03      1.195e+01
135          2.197e-03      1.239e+01
140          1.664e-03      1.283e+01
145          1.262e-03      1.327e+01
150          9.590e-04      1.370e+01
155          7.297e-04      1.413e+01
160          5.560e-04      1.456e+01
165          4.242e-04      1.499e+01
170          3.240e-04      1.547e+01
```

```
175          2.477e-04      1.591e+01
180          1.896e-04      1.634e+01
185          1.453e-04      1.677e+01
190          1.114e-04      1.720e+01
195          8.549e-05      1.763e+01
200          6.566e-05      1.806e+01
205          5.046e-05      1.850e+01
210          3.880e-05      1.893e+01
215          2.986e-05      1.937e+01
220          2.298e-05      1.981e+01
225          1.770e-05      2.025e+01
230          1.364e-05      2.070e+01
235          1.051e-05      2.112e+01
```

Let's have a look at the result

```
fig, ax = plt.subplots(figsize=(9, 5))
ax.set_ylim(-35, -24)
ax.plot(grid, v_star_approx, 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()
```

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 $\sigma(y) = (1 - \alpha\beta)y$

```
Tw, sigma = bellman_operator(v_star_approx,
                             grid,
                             beta,
                             np.log,
                             lambda k: k**alpha,
                             shocks,
                             compute_policy=True)


fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(grid, sigma, lw=2, alpha=0.6, label='approximate policy function')
cstar = (1 - alpha * beta) * grid
ax.plot(grid, cstar, lw=2, alpha=0.6, label='true policy function')
ax.legend(loc='lower right')
plt.show()
```

The figure shows that we've done a good job in this instance of approximating the true policy

Exercises

Exercise 1

Once an optimal consumption policy σ is given, the dynamics for the capital stock follows (5.15)

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

```
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

Solutions

Solution to Exercise 1

Here's our solution, which assumes you've executed the code in the lecture

```
def simulate_og(sigma, y0 = 0.1, ts_length=100):
    '''
    Compute a time series given consumption policy sigma.
    '''
    y = np.empty(ts_length)
    xi = np.random.randn(ts_length)
    y[0] = y0
    for t in range(ts_length-1):
        y[t+1] = (y[t] - sigma(y[t]))**alpha * np.exp(mu + s * xi[t+1])
    return y

fig, ax = plt.subplots(figsize=(9, 6))

for beta 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,
                                        beta,
                                        np.log,
                                        lambda k: k**alpha,
                                        shocks,
                                        Tw=Tw,
                                        compute_policy=False)

    Tw, sigma = bellman_operator(v_star_approx,
                             grid,
                             beta,
                             np.log,
                             lambda k: k**alpha,
                             shocks,
                             compute_policy=True)

    sigma_func = lambda x: np.interp(x, grid, sigma)
    y = simulate_og(sigma_func)
    ax.plot(y, lw=2, alpha=0.6, label=r'$\beta = {}$'.format(beta))


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

Time Iteration with Euler Equations

Contents

- *Time Iteration with Euler Equations*
  - *Overview*
  - *The Euler Equation*
  - *Comparison with Value Function Iteration*
  - *Implementation*
  - *Exercises*
  - *Solutions*

Overview

In this lecture we'll continue our *earlier study* of the stochastic optimal growth model

In that lecture we solved the associated discounted dynamic programming problem using value function iteration

The beauty of this technique is its broad applicability

With numerical problems, however, we can often attain higher efficiency in specific applications by deriving methods that are carefully tailored to the application at hand

The stochastic optimal growth model has plenty of structure to exploit for this purpose, especially when we adopt some concavity and smoothness assumptions over primitives

We'll use this structure to obtain an **Euler equation** based method that's more efficient than value function iteration for this and some other closely related applications

In a *subsequent lecture* we'll see that the numerical implementation part of the Euler equation method can be further adjusted to obtain even more efficiency

The Euler Equation

Let's take the model set out in *the stochastic growth model lecture* and add the assumptions that

1. u and f are continuously differentiable and strictly concave
2. $f(0) = 0$
3. $\lim_{c \to 0} u'(c) = \infty$ and $\lim_{c \to \infty} u'(c) = 0$
4. $\lim_{k \to 0} f'(k) = \infty$ and $\lim_{k \to \infty} f'(k) = 0$

The last two conditions are usually called **Inada conditions**

Recall the Bellman equation

$$v^*(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y-c)z)\phi(dz) \right\} \quad \text{for all} \quad y \in \mathbb{R}_+ \tag{5.23}$$

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 (5.23)

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

$$(v^*)'(y) = u'(c^*(y)) := (u' \circ c^*)(y) \tag{5.24}$$

The last result is called the **envelope condition** due to its relationship with the envelope theorem

To see why (5.24) 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 iteriority of the optimal policy imply that optimal consumption satisfies the first order condition associated with (5.23), which is

$$u'(c^*(y)) = \beta \int (v^*)'(f(y-c^*(y))z) f'(y-c^*(y)) z \phi(dz) \tag{5.25}$$

Combining (5.24) and the first-order condition (5.25) gives the famous **Euler equation**

$$(u' \circ c^*)(y) = \beta \int (u' \circ c^*)(f(y-c^*(y))z) f'(y-c^*(y)) z \phi(dz) \tag{5.26}$$

We can think of the Euler equation as a functional equation

$$(u' \circ \sigma)(y) = \beta \int (u' \circ \sigma)(f(y-\sigma(y))z) f'(y-\sigma(y)) z \phi(dz) \tag{5.27}$$

over interior consumption policies σ, one solution of which is the optimal policy c^*

Our aim is to solve the functional equation (5.27) and hence obtain c^*

The Coleman Operator

Recall the Bellman operator

$$Tw(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y-c)z)\phi(dz) \right\} \tag{5.28}$$

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 $\mathscr{P}$

1. The operator K takes as its argument a $\sigma \in \mathscr{P}$ and
2. returns a new function $K\sigma$, where $K\sigma(y)$ is the $c \in (0, y)$ that solves

$$u'(c) = \beta \int (u' \circ \sigma)(f(y-c)z) f'(y-c) z \phi(dz) \tag{5.29}$$

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 σ

The important thing to note about K is that, by construction, its fixed points coincide with solutions to the functional equation (5.27)

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 (5.29)?

The answer is yes, under our assumptions

For any $\sigma \in \mathscr{P}$, the right side of (5.29)

- is continuous and strictly increasing in c on $(0, y)$
- diverges to $+\infty$ as $c \uparrow y$

The left side of (5.29)

- 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 $K\sigma \in \mathscr{P}$ whenever $\sigma \in \mathscr{P}$

Comparison with Value Function Iteration

How does Euler equation time iteration compare with value function iteration?

Both can be used to compute the optimal policy, but is one faster or more accurate?

There are two parts to this story

First, on a theoretical level, the two methods are essentially isomorphic

In particular, they converge at the same rate

We'll prove this in just a moment

The other side 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

Let's talk about the theory first

To explain the connection between the two algorithms, it helps to understand the notion of equivalent dynamics

(This concept is very helpful in many other contexts as well)

Suppose that we have a function $g \colon X \to X$ where X is a given set

The pair (X, g) is sometimes called a **dynamical system** and we associate it with trajectories of the form

$$x_{t+1} = g(x_t), \qquad x_0 \text{ given}$$

Equivalently, $x_t = g^t(x_0)$, where g is the t-th composition of g with itself

Here's the picture

$$x_0 \xrightarrow{g} g(x_0) \xrightarrow{g} g^2(x_0) \xrightarrow{g} g^3(x_0) \xrightarrow{g} \cdots$$

Now let another function $h \colon Y \to Y$ where Y is another set

Suppose further that

- there exists a bijection τ from X to Y

- the two functions **commute** under τ, which is to say that $\tau(g(x)) = h(\tau(x))$ for all $x \in X$

The last statement can be written more simply as

$$\tau \circ g = h \circ \tau$$

or, by applying τ^{-1} to both sides

$$g = \tau^{-1} \circ h \circ \tau \tag{5.30}$$

Here's a commutative diagram that illustrates

$$\begin{array}{ccc} X & \xrightarrow{g} & X \\ \tau \big\downarrow & & \big\uparrow \tau^{-1} \\ Y & \xrightarrow[h]{} & Y \end{array}$$

Here's a similar figure that traces out the action of the maps on a point $x \in X$

$$\begin{array}{ccc} x & \xrightarrow{g} & g(x) \\ \tau \big\downarrow & & \big\uparrow \tau^{-1} \\ \tau(x) & \xrightarrow[h]{} & h(\tau(x)) \end{array}$$

Now, it's easy to check from (5.30) that $g^2 = \tau^{-1} \circ h^2 \circ \tau$ holds

In fact, if you like proofs by induction, you won't have trouble showing that

$$g^n = \tau^{-1} \circ h^n \circ \tau$$

is valid for all n

What does this tell us?

It tells us that the following are equivalent

- iterate n times with g, starting at x
- shift x to Y using τ, iterate n times with h starting at $\tau(x)$, and shift the result $h^n(\tau(x))$ back to X using τ^{-1}

We end up with exactly the same object

Back to Economics

Have you guessed where this is leading?

What we're going to show now is that the operators T and K commute under a certain bijection

The implication is that they have exactly the same rate of convergence

To make life a little easier, we'll assume in the following analysis (although not always in our applications) that $u(0) = 0$

A Bijection

Let $\mathscr{V}$ 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 $v \in \mathscr{V}$ let

$$Mv := h \circ v' \qquad \text{where } h := (u')^{-1}$$

Although we omit details, $\sigma := Mv$ is actually the unique v-greedy policy

- See proposition 12.1.18 of EDTC

It turns out that M is a bijection from $\mathscr{V}$ to $\mathscr{P}$

A (solved) exercise below asks you to confirm this

Commutative Operators

It is an additional solved exercise (see below) to show that T and K commute under M, in the sense that

$$M \circ T = K \circ M \tag{5.31}$$

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!

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 arises

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

```
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) *␣
↪shocks
            return u_prime(c) - beta * np.mean(vals)
```

```
        c_star = brentq(h, 1e-10, y - 1e-10)
        Kg[i] = c_star

    return Kg
```

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

Let's see that Bellman operator code again, since we'll use it in some tests below

```
import numpy as np
from scipy.optimize import fminbound


def bellman_operator(w, grid, beta, 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
    beta : 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:
        sigma = np.empty_like(w)

    # == set Tw[i] = max_c { u(c) + beta E w(f(y  - c) z)} == #
    for i, y in enumerate(grid):
        def objective(c):
            return - u(c) - beta * np.mean(w_func(f(y - c) * shocks))
        c_star = fminbound(objective, 1e-10, y)
        if compute_policy:
            sigma[i] = c_star
        Tw[i] = - objective(c_star)

    if compute_policy:
        return Tw, sigma
    else:
        return Tw
```

Testing on the Log / Cobb–Douglas case

As we *did for value function iteration*, let's start by testing our method in the presence of a model that does have an analytical solution

We assume the following imports

```
import matplotlib.pyplot as plt
import quantecon as qe
```

Now let's bring in the log-linear growth model 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 ~ LN(mu, s).

    The class holds parameters and true value and policy functions.
    """

    def __init__(self, alpha=0.4, beta=0.96, mu=0, s=0.1):

        self.alpha, self.beta, self.mu, self.s = alpha, beta, mu, s

        # == Some useful constants == #
        self.ab = alpha * beta
        self.c1 = np.log(1 - self.ab) / (1 - beta)
        self.c2 = (mu + alpha * np.log(self.ab)) / (1 - alpha)
        self.c3 = 1 / (1 - beta)
```

```
        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.alpha

    def f_prime(self, k):
        return self.alpha * k**(self.alpha - 1)

    def c_star(self, y):
        " True optimal policy.  "
        return (1 - self.alpha * self.beta) * 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 == #
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
```

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(mu + s * np.random.randn(shock_size))
```

As a preliminary test, let's see if $Kc^* = c^*$, as implied by the theory

```
c_star_new = coleman_operator(c_star(grid),
             grid, beta, 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:

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'll use is the one that eats the whole pie: $c(y) = y$

```
g = grid
n = 15
fig, ax = plt.subplots(figsize=(9, 6))
lb = 'initial condition $c(y) = y$'
ax.plot(grid, g, color=plt.cm.jet(0), lw=2, alpha=0.6, label=lb)
for i in range(n):
    new_g = coleman_operator(g, grid, beta, u_prime, f, f_prime, shocks)
    g = new_g
    ax.plot(grid, g, color=plt.cm.jet(i / n), lw=2, alpha=0.6)

lb = 'true policy function $c^*$'
ax.plot(grid, c_star(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

Now let's compare the accuracy of iteration using the Coleman and Bellman operators

We'll generate

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'll 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

```
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

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, beta, u, f, shocks)
```

```
    g, w = new_g, new_w

new_w, vf_g = bellman_operator(w, grid, beta, 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's 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

Exercises

Exercise 1

Show that (5.31) 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 $\mathscr{V}$ to $\mathscr{P}$

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

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

$$u'(c(y)) = \beta \int v'(f(y - c(y))z) f'(y - c(y)) z \phi(dz) \tag{5.32}$$

Hence $MTv(y) = (u')^{-1}(u'(c(y))) = c(y)$

On the other hand, $KMv(y)$ is the $c(y)$ that solves

$$\begin{aligned} u'(c(y)) &= \beta \int (u' \circ (Mv))(f(y - c(y))z) f'(y - c(y)) z \phi(dz) \\ &= \beta \int (u' \circ ((u')^{-1} \circ v'))(f(y - c(y))z) f'(y - c(y)) z \phi(dz) \\ &= \beta \int v'(f(y - c(y))z) f'(y - c(y)) z \phi(dz) \end{aligned}$$

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 $\mathscr{V}$ to $\mathscr{P}$

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 $v \in \mathscr{V}$, the derivative v' is a continuous, strictly decreasing function

Hence, for fixed $v \in \mathscr{V}$, 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 $\mathscr{P}$

To see that each $\sigma \in \mathscr{P}$ has a preimage $v \in \mathscr{V}$ with $Mv = \sigma$, fix any $\sigma \in \mathscr{P}$

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 $v \in \mathscr{V}$ and $Mv = \sigma$

It's also true that M is one-to-one on $\mathscr{V}$

To see this, suppose that v and w are elements of $\mathscr{V}$ 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

```
## 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)

def u(c):
    return (c**(1 - gamma) - 1) / (1 - gamma)

def u_prime(c):
    return c**(-gamma)

grid = np.linspace(grid_min, grid_max, grid_size)
shocks = np.exp(mu + s * np.random.randn(shock_size))

## Let's make convenience functions based around these primitives

def crra_bellman(w):
    return bellman_operator(w, grid, beta, u, f, shocks)

def crra_coleman(g):
    return coleman_operator(g, grid, beta, 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, beta, 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()
```

Here's the resulting figure

The policies are indeed close

Solution to Exercise 4

Here's the code

It assumes that you've just run the code from the previous exercise

```
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
qe.util.tic()
for i in range(sim_length):
    new_g = crra_coleman(g)
    g = new_g
qe.util.toc()
```

If you run this you'll 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

The Endogenous Grid Method

Contents

- *The Endogenous Grid Method*
  - *Overview*
  - *Key Idea*
  - *Implementation*
  - *Speed*

Overview

We solved the stochastic optimal growth model using

1. *value function iteration*
2. *Euler equation based time iteration*

We found time iteration to be significantly more accurate at each step

In this lecture we'll look at an ingenious twist on the time iteration technique called the **endogenous grid method** (EGM)

EGM is a numerical method for implementing policy iteration invented by Chris Carroll

It is a good example of how a clever algorithm can save a massive amount of computer time

(Massive when we multiply saved CPU cycles on each implementation times the number of implementations worldwide)

The original reference is *[Car06]*

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

$$(u' \circ c^*)(y) = \beta \int (u' \circ c^*)(f(y - c^*(y))z) f'(y - c^*(y)) z \phi(dz) \tag{5.33}$$

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

$$u'(c) = \beta \int (u' \circ g)(f(y - c)z) f'(y - c) z \phi(dz) \tag{5.34}$$

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 represention 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 (5.34) 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

$$c_i = (u')^{-1} \left\{ \beta \int (u' \circ g)(f(k_i)z) \, f'(k_i) \, z \, \phi(dz) \right\} \tag{5.35}$$

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 (5.34)

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**

Implementation

Let's implement this version of the Coleman operator and see how it performs

The Operator

Here's an implementation of K using EGM as described above

```
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

Compare this with our original implementation, repeated below for convenience, which uses an exogenous grid and requires root finding:

```
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) *␣
↪shocks
            return u_prime(c) - beta * 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*, let's 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 ~ LN(mu, s).

    The class holds parameters and true value and policy functions.
    """

    def __init__(self, alpha=0.4, beta=0.96, mu=0, s=0.1):

        self.alpha, self.beta, self.mu, self.s = alpha, beta, mu, s

        # == Some useful constants == #
```

```
        self.ab = alpha * beta
        self.c1 = np.log(1 - self.ab) / (1 - beta)
        self.c2 = (mu + alpha * np.log(self.ab)) / (1 - alpha)
        self.c3 = 1 / (1 - beta)
        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.alpha

    def f_prime(self, k):
        return self.alpha * k**(self.alpha - 1)

    def c_star(self, y):
        " True optimal policy.  "
        return (1 - self.alpha * self.beta) * 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 == #
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
```

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(mu + s * np.random.randn(shock_size))
```

As a preliminary test, let's see if $Kc^* = c^*$, as implied by the theory

```
c_star_new = coleman_egm(c_star,
            k_grid, beta, 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 we're 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_inv* are the same

In any case, here's the result:

We can't really distinguish the two plots

In fact it's easy to see that the difference is essentially zero:

```
max(abs(c_star_new(k_grid) - c_star(k_grid)))
```

```
4.4408920985006262e-16
```

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: $c(y) = y$

```
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, beta, 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

Speed

Now let's compare the clock times per iteration for the standard Coleman operator (with exogenous grid) and the EGM version

We'll do so using the CRRA model adopted in the exercises of the *Euler equation time iteration lecture*

Here's the model and some convenient functions

```
## 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(mu + 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)
```

Here's the result

```
## 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

The Income Fluctuation Problem

Contents

- *The Income Fluctuation Problem*
  - *Overview*
  - *The Optimal Savings Problem*
  - *Computation*
  - *Exercises*
  - *Solutions*

Overview

Next we study an optimal savings problem for an infinitely lived consumer—the "common ancestor" described in *[LS12]*, 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 *[LS12]*
- 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]*

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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$c_t + a_{t+1} \leq R a_t + z_t, \qquad c_t \geq 0, \qquad a_t \geq -b \qquad t = 0, 1, \ldots \tag{5.36}$$

Here

- $\beta \in (0, 1)$ is the discount factor
- a_t is asset holdings at time t, with ad-hoc borrowing constraint $a_t \geq -b$
- c_t is consumption
- z_t is non-capital income (wages, unemployment compensation, etc.)

- $R := 1 + r$, where $r > 0$ is the interest rate on savings

Non-capital income $\{z_t\}$ is assumed to be a Markov process taking values in $Z \subset (0, \infty)$ with stochastic kernel Π

This means that $\Pi(z, B)$ is the probability that $z_{t+1} \in B$ given $z_t = z$

The expectation of $f(z_{t+1})$ given $z_t = z$ is written as

$$\int f(\acute{z})\, \Pi(z, d\acute{z})$$

We further assume that

1. $r > 0$ and $\beta R < 1$
2. u is smooth, strictly increasing and strictly concave with $\lim_{c \to 0} u'(c) = \infty$ and $\lim_{c \to \infty} u'(c) = 0$

The asset space is $[-b, \infty)$ and the state is the pair $(a, z) \in S := [-b, \infty) \times Z$

A *feasible consumption path* from $(a, z) \in S$ is a consumption sequence $\{c_t\}$ such that $\{c_t\}$ and its induced asset path $\{a_t\}$ satisfy

1. $(a_0, z_0) = (a, z)$
2. the feasibility constraints in (5.36), 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* $V \colon S \to \mathbb{R}$ is defined by

$$V(a, z) := \sup\, \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\} \tag{5.37}$$

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 (5.37)

To pin down such paths we can use a version of the Euler equation, which in the present setting is

$$u'(c_t) \geq \beta R\, \mathbb{E}_t[u'(c_{t+1})] \tag{5.38}$$

and

$$u'(c_t) = \beta R\, \mathbb{E}_t[u'(c_{t+1})] \quad \text{whenever } c_t < Ra_t + z_t + b \tag{5.39}$$

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 (5.38) and (5.39) are equivalent to

$$u'(c_t) = \max\left\{\beta R \mathbb{E}_t[u'(c_{t+1})]\,,\; u'(Ra_t + z_t + b)\right\} \tag{5.40}$$

Optimality Results

Given our assumptions, it is known that

1. For each $(a, z) \in S$, a unique optimal consumption path from (a, z) exists
2. This path is the unique feasible path from (a, z) satisfying the Euler equality (5.40) and the transversality condition

$$\lim_{t\to\infty} \beta^t \,\mathbb{E}\left[u'(c_t)a_{t+1}\right] = 0. \tag{5.41}$$

Moreover, there exists an *optimal consumption function* $c^*\colon S \to [0, \infty)$ such that the path from (a, z) generated by

$$(a_0, z_0) = (a, z), \quad z_{t+1} \sim \Pi(z_t, dy), \quad c_t = c^*(a_t, z_t) \quad \text{and} \quad a_{t+1} = Ra_t + z_t - c_t$$

satisfies both (5.40) and (5.41), and hence is the unique optimal path from (a, z)

In summary, to solve the optimization problem, we need to compute c^*

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 (5.40) to make it a statement about functions rather than random variables

In particular, consider the functional equation

$$u' \circ c\,(a, z) = \max\left\{\gamma \int u' \circ c\,\{Ra + z - c(a, z),\; \acute{z}\}\,\Pi(z, d\acute{z})\,,\; u'(Ra + z + b)\right\} \tag{5.42}$$

where $\gamma := \beta R$ and $u' \circ c(s) := u'(c(s))$

Equation (5.42) is a functional equation in c

In order to identify a solution, let $\mathscr{C}$ be the set of candidate consumption functions $c\colon S \to \mathbb{R}$ such that

- each $c \in \mathscr{C}$ is continuous and (weakly) increasing
- $\min Z \leq c(a, z) \leq Ra + z + b$ for all $(a, z) \in S$

In addition, let $K \colon \mathscr{C} \to \mathscr{C}$ be defined as follows:

For given $c \in \mathscr{C}$, the value $Kc(a, z)$ is the unique $t \in J(a, z)$ that solves

$$u'(t) = \max \left\{ \gamma \int u' \circ c \left\{ Ra + z - t,\ \acute{z} \right\} \Pi(z, d\acute{z}) \,,\ u'(Ra + z + b) \right\} \tag{5.43}$$

where

$$J(a, z) := \{ t \in \mathbb{R} \ : \ \min Z \leq t \leq Ra + z + b \} \tag{5.44}$$

We refer to K as Coleman's policy function operator *[Col90]*

It is known that

- K is a contraction mapping on $\mathscr{C}$ under the metric
$$\rho(c, d) := \| \, u' \circ c - u' \circ d \, \| := \sup_{s \in S} | \, u'(c(s)) - u'(d(s)) \, | \qquad (c, d \in \mathscr{C})$$
- The metric ρ is complete on $\mathscr{C}$
- Convergence in ρ implies uniform convergence on compacts

In consequence, K has a unique fixed point $c^* \in \mathscr{C}$ and $K^n c \to c^*$ as $n \to \infty$ for any $c \in \mathscr{C}$

By the definition of K, the fixed points of K in $\mathscr{C}$ coincide with the solutions to (5.42) in $\mathscr{C}$

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 $c \in \mathscr{C}$ and iterating with the operator K defined in (5.43)

Value Function Iteration

The Bellman operator for this problem is given by

$$Tv(a, z) = \max_{0 \leq c \leq Ra + z + b} \left\{ u(c) + \beta \int v(Ra + z - c, \acute{z}) \Pi(z, d\acute{z}) \right\} \tag{5.45}$$

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's 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`, which generates suitable initial conditions for iteration

```
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
    beta : scalar(float), optional(default=0.96)
        The discount factor, must satisfy (1 + r) * beta < 1
    Pi : 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, beta, Pi, z_vals, b, u, du : see Parameters
    asset_grid : np.ndarray
        One dimensional grid for assets

    """

    def __init__(self,
            r=0.01,
            beta=0.96,
            Pi=((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 === #
    R, Pi, beta, du, b = cp.R, cp.Pi, cp.beta, cp.du, cp.b
    asset_grid, z_vals = cp.asset_grid, cp.z_vals
    z_size = len(z_vals)
    gamma = R * beta
    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):
```

```
                expectation = np.dot(du(cf(R * a + z - t)), Pi[i_z, :])
                return du(t) - max(gamma * expectation, du(R * a + z + b))
            Kc[i_a, i_z] = brentq(h, 1e-8, R * a + z + b)

    return Kc

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, beta, u, b = cp.R, cp.beta, 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 - beta)

    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

Intiution behind this fact was provided in *a previous lecture on time iteration*

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

```
cp = ConsumerProblem()
v, c = initialize(cp)
```

```
%timeit bellman_operator(v, cp)
```

```
10 loops, best of 3: 142 ms per loop
```

```
%timeit coleman_operator(c, cp)
```

```
10 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 let's consider the long run asset levels held by households

We'll take $r = 0.03$ and otherwise use default parameters

The following figure is a 45 degree diagram showing the law of motion for assets when consumption is optimal

The green line and blue line represent the function

$$a' = h(a, z) := Ra + z - c^*(a, z)$$

when income z takes its high and low values repectively

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: *[LS12]* 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

Solutions

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

Exercise 1

```
cp = ConsumerProblem()
K = 80

# Bellman iteration
V, c = initialize(cp)
print("Starting value function iteration")
for i in range(K):
    # print "Current iterate = " + str(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 "Current iterate = " + str(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

```
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=r'$r = %.3f$' % r_val)

ax.set_xlabel('asset level')
ax.set_ylabel('consumption (low income)')
ax.legend(loc='upper left')
plt.show()
```

Exercise 3

```
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)
    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=r'$b = %d$' % b)
    print("Finished iteration b=%i" % b)

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
```

A Problem that Stumped Milton Friedman

(and that Abraham Wald solved by inventing sequential analysis)

Contents

- *A Problem that Stumped Milton Friedman*

- *Overview*
- *Origin of the problem*
- *A dynamic programming approach*
- *Implementation*
- *Analysis*
- *Comparison with Neyman-Pearson formulation*

Co-authors: Chase Coleman

Overview

This lecture describes a statistical decision problem encountered by Milton Friedman and W. Allen Wallis during World War II when they were analysts at the U.S. Government's Statistical Research Group at Columbia University

This problem led Abraham Wald *[Wal47]* to formulate **sequential analysis**, an approach to statistical decision problems intimitately related to dynamic programming

In this lecture, we apply dynamic programming algorithms to Friedman and Wallis and Wald's problem

Key ideas in play will be:

- Bayes' Law
- Dynamic programming
- Type I and type II statistical errors
  - a type I error occurs when you reject a null hypothesis that is true
  - a type II error is when you accept a null hypothesis that is false
- Abraham Wald's **sequential probability ratio test**
- The **power** of a statistical test
- The **critical region** of a statistical test
- A **uniformly most powerful test**

Origin of the problem

On pages 137-139 of his 1998 book *Two Lucky People* with Rose Friedman *[FF98]*, Milton Friedman described a problem presented to him and Allen Wallis during World War II, when they worked at the US Government's Statistical Research Group at Columbia University

Let's listen to Milton Friedman tell us what happened

"In order to understand the story, it is necessary to have an idea of a simple statistical problem, and of the standard procedure for dealing with it. The actual problem out of which sequential analysis grew will serve.

The Navy has two alternative designs (say A and B) for a projectile. It wants to determine which is superior. To do so it undertakes a series of paired firings. On each round it assigns the value 1 or 0 to A accordingly as its performance is superior or 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]*

We'll formulate the problem using dynamic programming

A dynamic programming approach

The following presentation of the problem closely follows Dmitri Berskekas's treatment in **Dynamic Programming and Stochastic Control** *[Ber75]*

A decision maker 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 $x = x_0$[1]

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

The code to generate this figure can be found in beta_plots.py

[1] Because the decision maker believes that z_{k+1} is drawn from a mixture of two i.i.d. distributions, he does *not* believe that the sequence $[z_{k+1}, z_{k+2}, \ldots]$ is i.i.d. Instead, he believes that it is *exchangeable*. See *[Kre88]* chapter 11, for a discussion of exchangeability.

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 z‘s
- He decides that $x = x_1$ and draws no more z‘s
- He postpones deciding now and instead chooses to draw a z_{k+1}

Associated with these three actions, the decision maker can suffer three kinds of losses:

- A loss L_0 if he decides $x = x_0$ 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

As we'll see, this is indeed the correct form of the decision rule

The key problem is to determine the threshold values α, β, which will depend on the parameters listed above

You might like to pause at this point and try to predict the impact of a parameter such as c or L_0 on α or β

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

$$J(p) = \min \big\{(1-p)L_0,\ pL_1,\ c + \mathbb{E}[J(p')]\big\} \tag{5.46}$$

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

$$\begin{aligned} \text{accept } x = x_0 \text{ if } p \geq \alpha \\ \text{accept } x = x_1 \text{ if } p \leq \beta \\ \text{draw another } z \text{ if } \beta \leq p \leq \alpha \end{aligned}$$

Our aim is to compute the value function J, and from it the associated cutoffs α and β

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

Implementation

Let's code this problem up and solve it

To approximate the value function that solves Bellman equation (5.46), 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

```
"""
A first pass at solving Wald's sequential decision problem using dynamic
programming.

"""

import numpy as np
import scipy.stats as st
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
    tp1_dist = np.clip((p*f0) / (p*f0 + (1-p)*f1), 0, 1)

    # Evaluate the expectation
    EJ = curr_dist @ J(tp1_dist)

    return EJ

def expect_loss_cont(p, c, f0, f1, J):
    return c + EJ(p, f0, f1, J)


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) L0, p L1, 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

```
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
5            8.042e-02      4.868e-02
10           6.418e-04      8.845e-02
15           4.482e-06      1.280e-01
```

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

```
"""
Solving Wald's sequential decision problem using dynamic programming.  An
approach using classes.

"""

import numpy as np
import scipy.stats as st
import scipy.interpolate as interp
import quantecon as qe

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

    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)
        tp1_dist = self.bayes_update_all(p)

        # Evaluate the expectation
        EJ = curr_dist @ J(tp1_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)

        self.J = J
        return J

    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
        drv = qe.discrete_rv.DiscreteRV(f)

        # 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 = drv.draw()[0]
            t = t+1
            p = update_p(p, k)
            if p < lb:
                decision_made = True
                decision = 1
            elif p > ub:
                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
```

Now let's use our class to solve Bellman equation (5.46) and verify that it gives similar output

```
# 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

```
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
5            8.042e-02      4.754e-02
10           6.418e-04      8.886e-02
15           4.482e-06      1.305e-01
```

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

Analysis

Now that our routines are working, let's inspect the solutions

We'll start with the following baseline parameters

```
c = 1.25            # Cost of another draw
L0 = 25             # Cost of selecting x0 when x1 is true
L1 = 25             # Cost of selecting x1 when x0 is true
a0, b0 = 2.5, 2.0   # Parameters for f0 (beta distribution)
a1, b1 = 2.0, 2.5   # Parameters for f1 (beta distribution)
m = 25              # Size of grid
```

Here's a plot of some objects we'll 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 β or above α

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?

Here's the figure

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

Comparison with Neyman-Pearson formulation

For several reasons, it is useful to describe the theory underlying the test that Navy Captain G. S. Schuyler had been told to use and that led him to approach Milton Friedman and Allan Wallis to convey his conjecture that superior practical procedures existed

Evidently, the Navy had told Captail Schuyler to use what it knew to be a state-of-the-art Neyman-Pearson test

We'll rely on Abraham Wald's *[Wal47]* elegant summary of Neyman-Pearson theory

For our purposes, watch for there features of the setup:

- the assumption of a *fixed* sample size n
- the application of laws of large numbers, conditioned on alternative probability models, to interpret the probabilities α and β defined in the Neyman-Pearson theory

Recall that in the sequential analytic formulation above, that

- The sample size n is not fixed but rather an object to be chosen; technically n is a random variable
- The parameters β and α characterize cut-off rules used to determine n as a random variable
- Laws of large numbers make no appearances in the sequential construction

In chapter 1 of **Sequential Analysis** *[Wal47]* Abraham Wald summarizes the Neyman-Pearson approach to hypothesis testing

Wald frames the problem as making a decision about a probability distribution that is partially known

(You have to assume that *something* is already known in order to state a well posed problem. Usually, *something* means *a lot*.)

By limiting what is unknown, Wald uses the following simple structure to illustrate the main ideas.

- a decision maker wants to decide which of two distributions f_0, f_1 govern an 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 hypthothesis 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 possibsle 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 2 which are not included in part 1, part 2 is uniquely determined by part 1. Thus, choosing a test procedure is equivalent to determining a critical region.

Let's listen to Wald longer:

- As a basis for choosing among critical regions the following considerations have been advanced by Neyman and Pearson: In accepting or rejecting H_0 we may commit errors of two kinds. We commit an error of the first kind if we reject H_0 when it is true; we commit an error of the second kind if we accept H_0 when H_1 is true. After a particular critical region W has been chosen, the probability of committing an error of the first kind, as well as the probability of committing an error of the second kind is uniquely determined. The probability of committing an error of the first kind is equal to the probability, determined by the assumption that H_0 is true, that the observed sample will be included in the critical region W. The probability of committing an error of the second kind is equal to the probability, determined on the assumption that H_1 is true, that the probability will fall outside the critical region W. For any given critical region W we shall denote the probability of an error of the first kind by α and the probability of an error of the second kind by β.

Let's listen carefully to how Wald applies a law of large numbers to interpret α and β:

- The probabilities α and β have the following important practical interpretation: Suppose that we draw a large number of samples of size n. Let M be the number of such samples drawn. Suppose that for each of these M samples we reject H_0 if the sample is included in W and accept H_0 if the sample lies outside W. In this way we make M statements of rejection or acceptance. Some of these statements will in general be wrong. If H_0 is true and if M is large, the probability is nearly 1 (i.e., it is practically certain) that the proportion of wrong statements (i.e., the number of wrong statements divided by M) will be approximately α. If H_1 is true, the probability is nearly 1 that the proportion of wrong statements will be approximately β. Thus, we can say that in the long run [here Wald applies a law of large numbers by driving $M \to \infty$ (our comment, not Wald's)] the proportion of wrong statements will be α if $H_0$is true and β if H_1 is true.

The quantity α is called the *size* of the critical region, and the quantity $1-\beta$ is called the *power* of the critical region.

Wald notes that

- one critical region W is more desirable than another if it has smaller values of α and β. Although either α or β can be made arbitrarily small by a proper choice of the critical region W, it is possible to make both α and β arbitrarily small for a fixed value of n, i.e., a fixed sample size.

Wald summarizes Neyman and Pearson's setup as follows:

- Neyman and Pearson show that a region consisting of all samples $(z_1, z_2, \ldots, z_n)$ which satisfy the inequality

$$\frac{f_1(z_1)\cdots f_1(z_n)}{f_0(z_1)\cdots f_1(z_n)} \geq k$$

  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 α.

Wald goes on to discuss Neyman and Pearson's concept of *uniformly most powerful* test.

Here is how Wald introduces the notion of a sequential test

- A rule is given for making one of the following three decisions at any stage of the experiment (at the m th trial for each integral value of m): (1) to accept the hypothesis H , (2) to reject the hypothesis H , (3) to continue the experiment by making an additional observation. Thus, such a test procedure is carried out sequentially. On the basis of the first observation one of the aforementioned 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.

The Permanent Income Model: I

Contents

- *The Permanent Income Model: I*
  - *Overview*
  - *The Savings Problem*
  - *Alternative Representations*
  - *Two Classic Examples*
  - *Further Reading*
  - *Appendix: The Euler Equation*

Overview

This lecture describes a rational expectations version of the famous permanent income model of Friedman *[Fri56]*

Hall cast Friedman's model within a linear-quadratic setting *[Hal78]*

Like Hall, we formulate an infinite-horizon linear-quadratic savings problem

We use the model as a vehicle for illustrating

- alternative formulations of the *state* of a dynamic system
- the idea of *cointegration*
- impulse response functions
- the idea that changes in consumption are useful as predictors of movements in income

Background readings on the linear-quadratic-Gaussian permanent income model are Robert Hall's *[Hal78]* and chapter 2 of *[LS12]*

The Savings Problem

In this section we state and solve the savings and consumption problem faced by the consumer

Preliminaries

The discussion below requires a casual familiarity with martingales

A discrete time martingale is a stochastic process (i.e., a sequence of random variables) $\{X_t\}$ with finite mean and satisfying

$$\mathbb{E}_t[X_{t+1}] = X_t, \qquad t = 0, 1, 2, \ldots$$

Here $\mathbb{E}_t := \mathbb{E}[\cdot \mid \mathcal{F}_t]$ is a 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

$$\mathbb{E}_0 \left[\sum_{t=0}^{\infty} \beta^t u(c_t) \right] \tag{5.47}$$

where

- $\mathbb{E}_t$ is the mathematical expectation conditioned on the consumer's time t information
- c_t is time t consumption
- u is a strictly concave one-period utility function
- $\beta \in (0, 1)$ is a discount factor

The consumer maximizes (5.47) by choosing a consumption, borrowing plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t \quad t \geq 0 \tag{5.48}$$

Here

- y_t is an exogenous endowment process
- $r > 0$ is the risk-free 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*

$$\begin{aligned} z_{t+1} &= Az_t + Cw_{t+1} \\ y_t &= Uz_t \end{aligned} \tag{5.49}$$

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 $\rho(A) < 1/\beta$

- U is a selection vector that pins down y_t as a particular linear combination of the elements of z_t.

The restriction on $\rho(A)$ prevents income from growing so fast that some discounted geometric sums of some infinite sequences below become infinite

Regarding preferences, we assume the quadratic utility function

$$u(c_t) = -(c_t - \gamma)^2$$

where γ is a bliss level of consumption

Note: Along with this quadratic utility specification, we allow consumption to be negative. However, by choosing parameters appropriately, we can make the probability that the model generates negative consumption paths as low as desired.

Finally, we impose the *no Ponzi scheme* condition

$$\mathbb{E}_0 \left[\sum_{t=0}^{\infty} \beta^t b_t^2 \right] < \infty \tag{5.50}$$

This condition rules out an always-borrow scheme that would allow the household to enjoy bliss consumption forever

First Order Conditions

First-order conditions for maximizing (5.47) subject to (5.48) are

$$\mathbb{E}_t[u'(c_{t+1})] = u'(c_t), \qquad t = 0, 1, \ldots \tag{5.51}$$

These equations are also known as the *Euler equations* for the model

If you're not sure where they come from, you can find a proof sketch in the *appendix*

With our quadratic preference specification, (5.51) has the striking implication that consumption follows a martingale:

$$\mathbb{E}_t[c_{t+1}] = c_t \tag{5.52}$$

(In fact quadratic preferences are *necessary* for this conclusion[1])

One way to interpret (5.52) is that consumption will only change when "new information" about permanent income is revealed

These ideas will be clarified below

[1] A linear marginal utility is essential for deriving (5.52) from (5.51). Suppose instead that we had imposed the following more standard assumptions on the utility function: $u'(c) > 0, u''(c) < 0, u'''(c) > 0$ and required that $c \geq 0$. The Euler equation remains (5.51). But the fact that $u''' < 0$ implies via Jensen's inequality that $\mathbb{E}_t[u'(c_{t+1})] > u'(\mathbb{E}_t[c_{t+1}])$. This inequality together with (5.51) implies that $\mathbb{E}_t[c_{t+1}] > c_t$ (consumption is said to be a 'submartingale'), so that consumption stochastically diverges to $+\infty$. The consumer's savings also diverge to $+\infty$.

The Optimal Decision Rule

Now let's deduce the optimal decision rule[2]

Note: One way to solve the consumer's problem is to apply *dynamic programming* as in *this lecture*. We do this later. But first we use an alternative approach that is revealing and shows the work that dynamic programming does for us automatically

---

In doing so, we need to combine

1. the optimality condition (5.52)
2. the period-by-period budget constraint (5.48), and
3. the boundary condition (5.50)

To accomplish this, observe first that (5.50) implies $\lim_{t\to\infty} \beta^t b_{t+1} = 0$

Using this restriction on the debt path and solving (5.48) forward yields

$$b_t = \sum_{j=0}^{\infty} \beta^j (y_{t+j} - c_{t+j}) \tag{5.53}$$

Take conditional expectations on both sides of (5.53) and use the martingale property of consumption and the *law of iterated expectations* to deduce

$$b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - \frac{c_t}{1-\beta} \tag{5.54}$$

Expressed in terms of c_t we get

$$c_t = (1-\beta)\left[\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - b_t\right] = \frac{r}{1+r}\left[\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - b_t\right] \tag{5.55}$$

where the last equality uses $(1+r)\beta = 1$

These last two equations assert that consumption equals *economic income*

- *financial wealth* equals $-b_t$
- *non-financial* wealth equals $\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}]$
- A *marginal propensity to consume out of wealth* equals the interest factor $\frac{r}{1+r}$
- *economic income* equals
  - a constant marginal propensity to consume times the sum of nonfinancial wealth and financial wealth
  - the amount the household can consume while leaving its wealth intact

[2] An optimal decision rule is a map from current state into current actions—in this case, consumption

Reacting to the state

The *state* vector confronting the household at t is $\begin{bmatrix} b_t & z_t \end{bmatrix}$

Here

- z_t is an *exogenous* component, unaffected by household 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 household's future endowment

It seems likely that current decisions c_t and b_{t+1} should be expressible as functions of z_t and b_t

This is indeed the case

In fact, from *this discussion* we see that

$$\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] = \mathbb{E}_t \left[\sum_{j=0}^{\infty} \beta^j y_{t+j} \right] = U(I - \beta A)^{-1} z_t$$

Combining this with (5.55) gives

$$c_t = \frac{r}{1+r} \left[U(I - \beta A)^{-1} z_t - b_t \right] \tag{5.56}$$

Using this equality to eliminate c_t in the budget constraint (5.48) gives

$$\begin{aligned} b_{t+1} &= (1+r)(b_t + c_t - y_t) \\ &= (1+r)b_t + r[U(I - \beta A)^{-1} z_t - b_t] - (1+r)Uz_t \\ &= b_t + U[r(I - \beta A)^{-1} - (1+r)I]z_t \\ &= b_t + U(I - \beta A)^{-1}(A - I)z_t \end{aligned}$$

To get from the second last to the last expression in this chain of equalities is not trivial

Try using the fact that $(1+r)\beta = 1$ and $(I - \beta A)^{-1} = \sum_{j=0}^{\infty} \beta^j A^j$

We've now successfully written c_t and b_{t+1} as functions of b_t and z_t

A State-Space Representation

We can summarize our dynamics in the form of a linear state-space system governing consumption, debt and income:

$$\begin{aligned} z_{t+1} &= Az_t + Cw_{t+1} \\ b_{t+1} &= b_t + U[(I - \beta A)^{-1}(A - I)]z_t \\ y_t &= Uz_t \\ c_t &= (1-\beta)[U(I - \beta A)^{-1} z_t - b_t] \end{aligned} \tag{5.57}$$

To write this more succinctly, let

$$x_t = \begin{bmatrix} z_t \\ b_t \end{bmatrix}, \quad \tilde{A} = \begin{bmatrix} A & 0 \\ U(I - \beta A)^{-1}(A - I) & 1 \end{bmatrix}, \quad \tilde{C} = \begin{bmatrix} C \\ 0 \end{bmatrix}$$

and

$$\tilde{U} = \begin{bmatrix} U & 0 \\ (1-\beta)U(I-\beta A)^{-1} & -(1-\beta) \end{bmatrix}, \quad \tilde{y}_t = \begin{bmatrix} y_t \\ c_t \end{bmatrix}$$

Then we can express equation (5.57) as

$$\begin{aligned} x_{t+1} &= \tilde{A}x_t + \tilde{C}w_{t+1} \\ \tilde{y}_t &= \tilde{U}x_t \end{aligned} \tag{5.58}$$

We can use the following formulas from *state-space representation* to compute population mean $\mu_t = \mathbb{E}x_t$ and covariance $\Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)']$

$$\mu_{t+1} = \tilde{A}\mu_t \quad \text{with} \quad \mu_0 \text{ given} \tag{5.59}$$

$$\Sigma_{t+1} = \tilde{A}\Sigma_t\tilde{A}' + \tilde{C}\tilde{C}' \quad \text{with} \quad \Sigma_0 \text{ given} \tag{5.60}$$

We can then compute the mean and covariance of $\tilde{y}_t$ from

$$\begin{aligned} \mu_{y,t} &= \tilde{U}\mu_t \\ \Sigma_{y,t} &= \tilde{U}\Sigma_t\tilde{U}' \end{aligned} \tag{5.61}$$

A Simple Example with iid Income

To gain some preliminary intuition on the implications of (5.57), let's look at a highly stylized example where income is just iid

(Later examples will investigate more realistic income streams)

In particular, let $\{w_t\}_{t=1}^{\infty}$ be iid and scalar standard normal, and let

$$z_t = \begin{bmatrix} z_t^1 \\ 1 \end{bmatrix}, \quad A = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & \mu \end{bmatrix}, \quad C = \begin{bmatrix} \sigma \\ 0 \end{bmatrix}$$

Finally, let $b_0 = z_0^1 = 0$

Under these assumptions we have $y_t = \mu + \sigma w_t \sim N(\mu, \sigma^2)$

Further, if you work through the state space representation, you will see that

$$b_t = -\sigma \sum_{j=1}^{t-1} w_j$$

$$c_t = \mu + (1-\beta)\sigma \sum_{j=1}^{t} w_j$$

Thus income is iid and debt and consumption are both Gaussian random walks

Defining assets as $-b_t$, we see that assets are just the cumulative sum of unanticipated income prior to the present date

The next figure shows a typical realization with $r = 0.05$, $\mu = 1$ and $\sigma = 0.15$

Observe that consumption is considerably smoother than income

The figure below shows the consumption paths of 250 consumers with independent income streams

The code for these figures can be found in perm_inc_figs.py

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

Hall's Representation

Hall *[Hal78]* suggests a sharp way to summarize the implications of LQ permanent income theory

First, to represent the solution for b_t, shift (5.55) forward one period and eliminate b_{t+1} by using (5.48) to obtain

$$c_{t+1} = (1-\beta)\sum_{j=0}^{\infty}\beta^j \mathbb{E}_{t+1}[y_{t+j+1}] - (1-\beta)\left[\beta^{-1}(c_t + b_t - y_t)\right]$$

If we add and subtract $\beta^{-1}(1-\beta)\sum_{j=0}^{\infty}\beta^j \mathbb{E}_t y_{t+j}$ from the right side of the preceding equation and rearrange, we obtain

$$c_{t+1} - c_t = (1-\beta)\sum_{j=0}^{\infty}\beta^j \left\{\mathbb{E}_{t+1}[y_{t+j+1}] - \mathbb{E}_t[y_{t+j+1}]\right\} \tag{5.62}$$

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 (5.62) and (5.54), which is repeated here:

$$b_t = \sum_{j=0}^{\infty}\beta^j \mathbb{E}_t[y_{t+j}] - \frac{1}{1-\beta}c_t \tag{5.63}$$

Equation (5.63) asserts that the household's debt due at t equals the expected present value of its endowment minus the expected present value of its consumption stream

A high debt thus indicates a large expected present value of surpluses $y_t - c_t$

Recalling again our discussion on *forecasting geometric sums*, we have

$$\begin{aligned}
\mathbb{E}_t \sum_{j=0}^{\infty}\beta^j y_{t+j} &= U(I-\beta A)^{-1} z_t \\
\mathbb{E}_{t+1} \sum_{j=0}^{\infty}\beta^j y_{t+j+1} &= U(I-\beta A)^{-1} z_{t+1} \\
\mathbb{E}_t \sum_{j=0}^{\infty}\beta^j y_{t+j+1} &= U(I-\beta A)^{-1} A z_t
\end{aligned}$$

Using these formulas together with (5.49) and substituting into (5.62) and (5.63) gives the following representation for the consumer's optimum decision rule:

$$\begin{aligned} c_{t+1} &= c_t + (1-\beta)U(I-\beta A)^{-1}Cw_{t+1} \\ b_t &= U(I-\beta A)^{-1}z_t - \frac{1}{1-\beta}c_t \\ y_t &= Uz_t \\ z_{t+1} &= Az_t + Cw_{t+1} \end{aligned} \tag{5.64}$$

Representation (5.64) makes clear that

- The state can be taken as (c_t, z_t)
  - The endogenous part is c_t and the exogenous part is z_t
  - Debt b_t has disappeared as a component of the state because it is encoded in c_t
- Consumption is a random walk with innovation $(1-\beta)U(I-\beta A)^{-1}Cw_{t+1}$
  - This is a more explicit representation of the martingale result in (5.52)

Cointegration

Representation (5.64) 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 processes to (certain transformations of) nonstationary models

To clarify 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 (5.57) 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 (5.64) we have

$$(1-\beta)b_t + c_t = (1-\beta)U(I-\beta A)^{-1}z_t \tag{5.65}$$

Hence the linear combination $(1-\beta)b_t + c_t$ is asymptotically stationary

Accordingly, Granger and Engle would call $\begin{bmatrix}(1-\beta) & 1\end{bmatrix}$ a *cointegrating vector* for the state

When applied to the nonstationary vector process $\begin{bmatrix}b_t & c_t\end{bmatrix}'$, it yields a process that is asymptotically stationary

Equation (5.65) can be arranged to take the form

$$(1-\beta)b_t + c_t = (1-\beta)\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j}, \tag{5.66}$$

Equation (5.66) asserts that the *cointegrating residual* on the left side equals the conditional expectation of the geometric sum of future incomes on the right[6]

[4] This would be the case if, for example, the *spectral radius* of A is strictly less than one

[6] See *[JYC88]*, *[LL01]*, *[LL04]* for interesting applications of related ideas.

Cross-Sectional Implications

Consider again (5.64), this time in light of our discussion of distribution dynamics in the *lecture on linear systems*

The dynamics of c_t are given by

$$c_{t+1} = c_t + (1-\beta)U(I-\beta A)^{-1}Cw_{t+1} \tag{5.67}$$

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

$$\mathrm{Var}[c_t] = \mathrm{Var}[c_0] + t\,\hat{\sigma}^2 \tag{5.68}$$

when

$$\hat{\sigma}^2 := (1-\beta)^2 U(I-\beta A)^{-1}CC'(I-\beta A')^{-1}U'$$

Assuming that $\hat{\sigma} > 0$, this means that $\{c_t\}$ has no asymptotic distribution

Let's consider what this means for a cross-section of ex ante identical households born at time 0

Let the distribution of c_0 represent the cross-section of initial consumption values

Equation (5.68) tells us that the distribution of c_t spreads out over time at a rate proportional to t

A number of different studies have investigated this prediction (see, e.g., *[DP94]*, *[STY04]*)

Impulse Response Functions

Impulse response functions measure the change in a dynamic system subject to a given impulse (i.e., temporary shock)

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 (5.49) has the moving average representation

$$y_{t+1} = d(L)w_{t+1} \tag{5.69}$$

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 household 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

$$\mathbb{E}_{t+1}[y_{t+j}] - \mathbb{E}_t[y_{t+j}] = d_{j-1} w_{t+1} \tag{5.70}$$

Using (5.70) in (5.62) gives

$$c_{t+1} - c_t = (1-\beta) d(\beta) w_{t+1} \tag{5.71}$$

The object $d(\beta)$ is the *present value of the moving average coefficients* in the representation for the endowment process y_t

Two Classic Examples

We illustrate some of the preceding ideas with the following two examples

In both examples, the endowment follows the process $y_t = z_{1t} + z_{2t}$ where

$$\begin{bmatrix} z_{1t+1} \\ z_{2t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} z_{1t} \\ z_{2t} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} w_{1t+1} \\ w_{2t+1} \end{bmatrix}$$

Here

- w_{t+1} is an iid 2×1 process distributed as $N(0, I)$
- z_{1t} is a permanent component of y_t
- z_{2t} is a purely transitory component

Example 1

Assume as before that the consumer observes the state z_t at time t

In view of (5.64) we have

$$c_{t+1} - c_t = \sigma_1 w_{1t+1} + (1-\beta)\sigma_2 w_{2t+1} \tag{5.72}$$

Formula (5.72) shows how an increment $\sigma_1 w_{1t+1}$ to the permanent component of income z_{1t+1} leads to

- a permanent one-for-one increase in consumption and
- no increase in savings $-b_{t+1}$

[3] Representation (5.49) implies that $d(L) = U(I - AL)^{-1}C$.

[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.

But the purely transitory component of income $\sigma_2 w_{2t+1}$ leads to a permanent increment in consumption by a fraction $1-\beta$ of transitory income

The remaining fraction β is saved, leading to a permanent increment in $-b_{t+1}$

Application of the formula for debt in (5.57) to this example shows that

$$b_{t+1} - b_t = -z_{2t} = -\sigma_2 w_{2t} \tag{5.73}$$

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

The code for generating this figure is in file `perm_inc_ir.py`, and can be downloaded here, as shown below

```
"""
Impulse response functions for the LQ permanent income model permanent and
transitory shocks.
"""
```

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

r       = 0.05
beta    = 1 / (1 + r)
T       = 20  # Time horizon
S       = 5   # Impulse date
sigma1  = sigma2 = 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] - sigma2 * w2[t]
        c[t+1] = c[t] + sigma1 * w1[t+1] + (1 - beta) * sigma2 * w2[t+1]
    return b, c


fig, axes = plt.subplots(2, 1)
plt.subplots_adjust(hspace=0.5)
p_args = {'lw': 2, 'alpha': 0.7}

L = 0.175

for ax in axes:
    ax.grid(alpha=0.5)
    ax.set_xlabel(r'Time')
    ax.set_ylim(-L, L)
    ax.plot((S, S), (-L, L), 'k-', lw=0.5)

ax = axes[0]
b, c = time_path(permanent=0)
ax.set_title('impulse-response, transitory 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.legend(loc='upper right')

ax = axes[1]
b, c = time_path(permanent=1)
ax.set_title('impulse-response, permanent 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.legend(loc='lower right')
plt.show()
```

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 (5.64)

The discussion in sections 2.9.1 and 2.11.3 of *[LS12]* shows that the pertinent state space representation for y_t is

$$\begin{bmatrix} y_{t+1} \\ a_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & -(1-K) \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ a_t \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} a_{t+1}$$
$$y_t = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ a_t \end{bmatrix}$$

where

- $K :=$ the stationary Kalman gain
- $a_t := y_t - E[y_t \,|\, y_{t-1}, \ldots, y_0]$

In the same discussion in *[LS12]* it is shown that $K \in [0, 1]$ and that K increases as σ_1/σ_2 does

In other words, as the ratio of the standard deviation of the permanent shock to that of the transitory shock increases

Applying formulas (5.64) implies

$$c_{t+1} - c_t = [1 - \beta(1-K)]a_{t+1} \tag{5.74}$$

where the endowment process can now be represented in terms of the univariate innovation to y_t as

$$y_{t+1} - y_t = a_{t+1} - (1-K)a_t \tag{5.75}$$

Equation (5.75) 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 (5.75), the first difference of income is a first-order moving average

Equation (5.74) asserts that the first difference of consumption is iid

Application of formula to this example shows that

$$b_{t+1} - b_t = (K-1)a_t \tag{5.76}$$

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

Further Reading

The model described above significantly changed how economists think about consumption

At the same time, it's generally recognized that Hall's version of the permanent income hypothesis fails to capture all aspects of the consumption/savings data

For example, liquidity constraints and buffer stock savings appear to be important

Further discussion can be found in, e.g., *[HM82]*, *[Par99]*, *[Dea91]*, *[Car01]*

Appendix: The Euler Equation

Where does the first order condition (5.51) come from?

Here we'll give a proof for the two period case, which is representative of the general argument

The finite horizon equivalent of the no-Ponzi condition is that the agent cannot end her life in debt, so $b_2 = 0$

From the budget constraint (5.48) 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

Subsituting these constraints into our two period objective $u(c_0) + \beta \mathbb{E}_0[u(c_1)]$ gives

$$\max_{b_1} \left\{ u\left(\frac{b_1}{R} - b_0 + y_0\right) + \beta \, \mathbb{E}_0[u(y_1 - b_1)] \right\}$$

You will be able to verify that the first order condition is

$$u'(c_0) = \beta R \, \mathbb{E}_0[u'(c_1)]$$

Using $\beta R = 1$ gives (5.51) in the two period case

The proof for the general case is similar

The Permanent Income Model: II

Contents

- *The Permanent Income Model: II*
  - *Overview*
  - *Setup*
  - *The LQ Approach*
  - *Implementation*
  - *Two Example Economies*

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 aggregate nondurable consumption and nonfinancial income

For example, we saw how the model asserts that

- consumption is a random walk
- financial wealth has a unit root and is cointegrated with consumption

It's easy to find other applications that show the usefulness of 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 tax collections and its expenditures net of debt service

This isomorphism means that in analyzing the LQ permanent income model, we are in effect also analyzing properties of the Barro tax smoothing model

In this lecture, we'll

- show how the solution to the LQ permanent income model can be obtained using LQ control methods
- represent the model as a linear state space system, and
- apply QuantEcon's LinearStateSpace class to characterize statistical features of the consumer's optimal consumption and borrowing plans

We'll then use these characterizations to construct a simple model of cross-section wealth and consumption dynamics in the spirit of Truman Bewley *[Bew86]*

(Later we'll study other Bewley models—see *this lecture*)

The model will prove useful for illustrating concepts such as

- stationarity
- ergodicity
- ensemble moments and cross section observations

Accompanying this lecture is a Jupyter notebook containing code and figures

This lecture is coauthored with Chase Coleman

Setup

Let's recall the basic features of the model as discussed in the *previous permanent income model lecture*

Consumer preferences are given by

$$E_0 \sum_{t=0}^{\infty} \beta^t u(c_t), \tag{5.77}$$

where $u(c) = -(c-\gamma)^2$

The consumer maximizes (5.77) by choosing a consumption, borrowing plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t, \quad t \geq 0 \tag{5.78}$$

and the no-Ponzi condition

$$E_0 \sum_{t=0}^{\infty} \beta^t b_t^2 < \infty \tag{5.79}$$

The interpretation of all variables and parameters are the same as in the *previous lecture*

We continue to assume that $(1+r)\beta = 1$

The dynamics of $\{y_t\}$ again follow the linear state space model

$$\begin{aligned} z_{t+1} &= Az_t + Cw_{t+1} \\ y_t &= Uz_t \end{aligned} \tag{5.80}$$

The restrictions on the shock process and parameters are the same

Digression on a useful isomorphism

The LQ permanent income model of consumption is mathematically isomorphic with a version of Barro's *[Bar79]* model of tax smoothing.

In the LQ permanent income model

- the household faces an exogenous process of nonfinancial income
- the household wants to smooth consumption across states and time

In the Barro model

- a government faces an exogenous sequence of government purchases (excluding interest payments on its debt)
- a government wants to smooth tax collections across states and time

If we set

- T_t, total tax collections in Barro's model to consumption c_t in the LQ permanent income model
- G_t, exogenous government expenditures in Barro's model to nonfinancial incomce y_t in the permanent income model
- B_t, government risk-free one-period debt in Barro's model to risk-free one period debt in the LQ permanent income model

- R, the gross rate of return on risk-free one-period government debt in Barro's model to the gross rate of return 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

A specification of the state 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 (5.80), as discussed in our *lecture on linear models*

To do so we take

$$z_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ \sigma \\ 0 \end{bmatrix}, \quad \text{and} \quad U = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

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?

Our answer is that by doing so we gather insights about the structure of the model

Our earlier approach based on solving a system of expectational difference equations brought to the fore the role of the consumer's expectations about future nonfinancial income

On the other hand, formulating the model in terms of an LQ dynamic programming problem reminds us that

- finding the state (of a dynamic programming problem) is an art, and
- iterations on a Bellman equation implicitly jointly solve a forecasting problem and a control problem

The LQ Problem

Recall from our *lecture on LQ theory* that the optimal linear regulator problem is to choose a decision rule for u_t to minimize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t \{x_t' R x_t + u_t' Q u_t\},$$

subject to x_0 given, and the law of motion

$$x_{t+1} = A_0 x_t + B_0 u_t + C_0 w_{t+1}, \qquad t \geq 0, \tag{5.81}$$

where w_{t+1} is iid with mean vector zero and $\mathbb{E} w_t w_t' = I$

The subscripts in A_0, B_0, C_0 are to avoid clashing with notation in (5.80)

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 $d = \beta(1-\beta)^{-1}\text{trace}(PCC')$

The optimal policy is $u_t = -Fx_t$, where $F := \beta(Q + \beta B'PB)^{-1}B'PA$

Under an optimal decision rule F, the state vector x_t evolves according to $x_{t+1} = (A - BF)x_t + Cw_{t+1}$

Mapping into the LQ framework

To map into the LQ framework, we'll use

$$x_t := \begin{bmatrix} z_t \\ b_t \end{bmatrix} = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \\ b_t \end{bmatrix}$$

as the state vector and $u_t := c_t - \gamma$ as the control

With this notation and $U_\gamma := \begin{bmatrix} \gamma & 0 & 0 \end{bmatrix}$, we can write the state dynamics as in (5.81) when

$$A_0 := \begin{bmatrix} A & 0 \\ (1+r)(U_\gamma - U) & 1+r \end{bmatrix} \quad B_0 := \begin{bmatrix} 0 \\ 1+r \end{bmatrix} \quad \text{and} \quad C_0 := \begin{bmatrix} C \\ 0 \end{bmatrix} w_{t+1}$$

Please confirm for yourself that, with these definitions, the LQ dynamics (5.81) 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 4×4 matrix of zeros

However, there is one problem remaining

We have no direct way to capture the non-recursive restriction (5.79) on the debt sequence $\{b_t\}$ from within the LQ framework

To try to enforce it, we're going to use a trick: put a small penalty on b_t^2 in the criterion function

In the present setting, this means adding a small entry $\epsilon > 0$ in the $(4,4)$ position of R

That will induce a (hopefully) small approximation error in the decision rule

We'll check whether it really is small numerically soon

Implementation

Let's write some code to solve the model

The code can be executed sequential in Jupyter notebook cells or in a script

We assume the following imports

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

One comment before we start is that the bliss level of consumption γ in the utility function has no effect on the optimal decision rule

We saw this in the *previous premanent income lecture*

The reason is that it drops out of the Euler equation for consumption

In what follows we set it equal to unity

The exogeneous income process

First we create the objects for the optimal linear regulator

```
# 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.]])
C = np.array([[0.], [sigma], [0.]])
G = np.array([[0., 1., 0.]])

# Form LinearStateSpace system and pull off steady state moments
mu_z0 = np.array([[1.0], [0.0], [0.0]])
sig_z0 = np.zeros((3, 3))
Lz = qe.LinearStateSpace(A, C, G, mu_0=mu_z0, Sigma_0=sig_z0)
muz, muy, Sigz, Sigy = Lz.stationary_distributions()

# Mean vector of state for the savings problem
mxo = np.vstack([muz, 0.0])

# Create stationary covariance matrix of x -- start everyone off at b=0
a1 = np.zeros((3, 1))
aa = np.hstack([Sigz, a1])
bb = np.zeros((1, 4))
sxo = np.vstack([aa, bb])

# These choices will initialize the state vector of an individual at zero debt
# and the ergodic distribution of the endowment process. Use these to create
```

```
# the Bewley economy.
mxbewley = mxo
sxbewley = sxo
```

The next step is to create the matrices for the LQ system

```
A12 = np.zeros((3,1))
ALQ_l = np.hstack([A, A12])
ALQ_r = np.array([[0, -R, 0, R]])
ALQ = np.vstack([ALQ_l, ALQ_r])

RLQ = np.array([[0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 1e-9]])

QLQ = np.array([1.0])
BLQ = np.array([0., 0., 0., R]).reshape(4,1)
CLQ = np.array([0., sigma, 0., 0.]).reshape(4,1)
betaLQ = beta
```

Let's print these out and have a look at them

```
print("A = \n", ALQ)
print("B = \n", BLQ)
print("R = \n", RLQ)
print("Q = \n", 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 =
 [ 1.]
```

Now create the appropriate instance of an LQ model

```
LQPI = qe.LQ(QLQ, RLQ, ALQ, BLQ, C=CLQ, beta=betaLQ)
```

We'll save the implied optimal policy function soon compare them with what we get by employing an alternative solution method

```
P, F, d = LQPI.stationary_values()  # Compute value function and decision rule
ABF = ALQ - np.dot(BLQ,F)  #  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 complete a system of expectational linear difference equations
- and then solving those equations to obtain the solution

Expressed in state space notation, the solution took the form

$$
\begin{aligned}
z_{t+1} &= Az_t + Cw_{t+1} \\
b_{t+1} &= b_t + U[(I - \beta A)^{-1}(A - I)]z_t \\
y_t &= Uz_t \\
c_t &= (1-\beta)[U(I-\beta A)^{-1}z_t - b_t]
\end{aligned}
$$

Now we'll apply the formulas in this system

```
# Use the above formulas to create the optimal policies for b_{t+1} and c_t
b_pol = np.dot(G, la.inv(np.eye(3, 3) - beta*A)).dot(A - np.eye(3, 3))
c_pol = (1 - beta)*np.dot(G, la.inv(np.eye(3, 3) - beta*A))

#Create the A matrix for a LinearStateSpace instance
A_LSS1 = np.vstack([A, b_pol])
A_LSS2 = np.eye(4, 1, -3)
A_LSS = np.hstack([A_LSS1, A_LSS2])

# Create the C matrix for LSS methods
C_LSS = np.vstack([C, np.zeros(1)])

# Create the G matrix for LSS methods
G_LSS1 = np.vstack([G, c_pol])
G_LSS2 = np.vstack([np.zeros(1), -(1 - beta)])
G_LSS = np.hstack([G_LSS1, G_LSS2])

# Use the following values to start everyone off at b=0, initial incomes zero
mu_0 = np.array([1., 0., 0., 0.])
sigma_0 = np.zeros((4, 4))
```

A_LSS calculated as we have here should equal *ABF* calculated above using the LQ model

```
ABF - A_LSS
```

```
array([[ 0.    ,  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*

```
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 let's create instances of the LinearStateSpace class and use it to do some interesting experiments

To do this, we'll use the outcomes from our second method

Two Example Economies

In the spirit of Bewley models *[Bew86]*, we'll generate panels of consumers

We'll study two examples

- In each of these examples, there is a continuum of *ex ante* identical consumers

The examples differ only in the initial states with which we endow the consumers

All other parameter values are kept the same in the two examples

- In the first example, all consumers begin with zero nonfinancial income and zero debt
  - The consumers are thus *ex ante* identical
- In the second example, while all begin with zero debt, we draw their initial income levels from the invariant distribution of financial income
  - Consumers are *ex ante* heterogeneous

In the first example, consumers' nonfinancial income paths display prounounced transients early in the sample that will affect outcomes in striking ways

Those transient effects will not be present in the second example

We use methods that the LinearStateSpace class contains to simulate the model with our first set of intitial conditions

- 25 paths of the exogenous non-financial income process and the associated consumption and debt paths.

In the first set of graphs, the darker lines depict one particular sample path, while the lighter lines indicate the other 24 paths

- A second graph that 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=mu_0, Sigma_0=sigma_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

```
def income_consumption_debt_series(A, C, G, m0, s0, T=150, npaths=25):
    """
    This function takes initial conditions (m0, s0) and uses the Linear State␣
↪Space
    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=m0, Sigma_0=s0)

    # 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):
        mu_x, mu_y, sig_x, sig_y = next(moment_generator)
        cons_mean[t], cons_var[t] = mu_y[1], sig_y[1, 1]
        debt_mean[t], debt_var[t] = mu_x[3], sig_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_xlabel("t")
    ax[0].set_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")
    ax[1].set_ylabel("debt")

    fig.suptitle("Nonfinancial Income, Consumption, and Debt")

    return fig

def consumption_debt_fanchart(csim, cons_mean, cons_var,
                              bsim, debt_mean, debt_var):
    # Get T
    T =  bsim.shape[1]

    # Create Percentiles of cross-section distributions
    cmean = np.mean(cons_mean)
    c90 = 1.65*np.sqrt(cons_var)
    c95 = 1.96*np.sqrt(cons_var)
    c_perc_95p, c_perc_95m = cons_mean + c95, cons_mean - c95
    c_perc_90p, c_perc_90m = cons_mean + c90, cons_mean - c90

    # Create Percentiles of cross-section distributions
    dmean = np.mean(debt_mean)
    d90 = 1.65*np.sqrt(debt_var)
    d95 = 1.96*np.sqrt(debt_var)
    d_perc_95p, d_perc_95m = debt_mean + d95, debt_mean - d95
    d_perc_90p, d_perc_90m = debt_mean + d90, debt_mean - d90


    # Create second figure
    fig2, ax2 = plt.subplots(2, 1, figsize=(10, 8))
    xvals = np.arange(T)
```

```
    # Consumption fan
    ax2[0].plot(xvals, cons_mean, color="k")
    ax2[0].plot(csim.T, color="k", alpha=.25)
    ax2[0].fill_between(xvals, c_perc_95m, c_perc_95p, alpha=.25, color="b")
    ax2[0].fill_between(xvals, c_perc_90m, c_perc_90p, alpha=.25, color="r")
    ax2[0].set_ylim((cmean-15, cmean+15))
    ax2[0].set_ylabel("consumption")

    # Debt fan
    ax2[1].plot(xvals, debt_mean, color="k")
    ax2[1].plot(bsim.T, color="k", alpha=.25)
    ax2[1].fill_between(xvals, d_perc_95m, d_perc_95p, alpha=.25, color="b")
    ax2[1].fill_between(xvals, d_perc_90m, d_perc_90p, alpha=.25, color="r")
#     ax2[1].set_ylim()
    ax2[1].set_ylabel("debt")

    fig2.suptitle("Consumption/Debt over time")
    ax2[1].set_xlabel("t")


    return fig2
```

Now let's create figures with initial conditions of zero for y_0 and b_0

```
out = income_consumption_debt_series(A_LSS, C_LSS, G_LSS, mu_0, sigma_0)
bsim0, csim0, ysim0 = out[:3]
cons_mean0, cons_var0, debt_mean0, debt_var0 = out[3:]

fig_0 = consumption_income_debt_figure(bsim0, csim0, ysim0)
fig_02 = consumption_debt_fanchart(csim0, cons_mean0, cons_var0,
                                   bsim0, debt_mean0, debt_var0)

fig_0.show()
fig_02.show()
```

Here are the two figures

Nonfinancial Income, Consumption, and Debt
y and c
t
c
y
debt
t
b

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 that we can represent the optimal decision rule for consumption in terms of the **co-integrating relationship**

$$(1-\beta)b_t + c_t = (1-\beta)E_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \tag{5.82}$$

So at time 0 we have

$$c_0 = (1-\beta)E_0 \sum_{t=0}^{\infty} \beta^j y_t$$

This tells us that consumption starts at the value of an annuity from the expected discounted value of nonfinancial income

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 income

He uses the gap between consumption and income mostly to service the interest payments due on his debt

Thus, when we look at the panel of debt in the accompanying graph, we see that this is a group of *ex ante* indentical people each of whom starts with zero debt

All of them accumulate debt in anticipation of rising nonfinancial income

They expect their nonfinancial income to rise toward the invariant distribution of income, a consequence of our having started them at $y_{-1} = y_{-2} = 0$

Cointegration residual

The following figure plots realizations of the left side of (5.82), which, *as discussed in our last lecture*, is called the **cointegrating residual**

The right side can be thought of as an annuity payment on the expected present value of future income $E_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$

Early along a realization, c_t is approximately constant while $(1-\beta)b_t$ and $(1-\beta)E_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$ both rise markedly as the household's present value of income and borrowing rise pretty much together

This example illustrates the following point: the definition of cointegration implies that the cointegrating residual is *asymptotically* covariance stationary, not *covariance stationary*

The cointegrating residual for the specification with zero income and zero debt initially has a notable transient component that dominates its behavior early in the sample.

By specifying different initial conditions, we shall remove this transient in our second example to be presented below

```
def cointegration_figure(bsim, csim):
    """
    Plots the cointegration
    """
    # Create figure
    fig, ax = plt.subplots(figsize=(10, 8))
    ax.plot((1-beta)*bsim[0, :] + csim[0, :], color="k")
    ax.plot((1-beta)*bsim.T + csim.T, color="k", alpha=.1)

    fig.suptitle("Cointegration of Assets and Consumption")
    ax.set_xlabel("t")
    ax.set_ylabel("")

    return fig
```

```
fig = cointegration_figure(bsim0, csim0)
fig.show()
```

Here's the resulting figure

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 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 in anticipation of rising incomes

So with the economic primitives set as above, the economy converges to a steady state in which there is an excess aggregate supply of risk-free loans at a gross interest rate of R

This excess supply is filled by "foreigner lenders" willing to make those loans

We can use virtually the same code to rig a "poor man's Bewley *[Bew86]* model" in the following way

- as before, we start everyone at $b_0 = 0$
- But instead of starting everyone at $y_{-1} = y_{-2} = 0$, we draw $\begin{bmatrix} y_{-1} \\ y_{-2} \end{bmatrix}$ from the invariant distribution of the $\{y_t\}$ process

This rigs a closed economy in which people are borrowing and lending with each other at a gross risk-free interest rate of $R = \beta^{-1}$

Across the group of people being analyzed, risk-free loans are in zero excess supply

We have arranged primitives so that $R = \beta^{-1}$ clears the market for risk-free loans at zero aggregate excess supply

So the risk-free loans are being made from one person to another within our closed set of agent

There is no need for foreigners to lend to our group

Let's have a look at the corresponding figures

```
out = income_consumption_debt_series(A_LSS, C_LSS, G_LSS, mxbewley, sxbewley)
bsimb, csimb, ysimb = out[:3]
cons_meanb, cons_varb, debt_meanb, debt_varb = out[3:]

fig_0 = consumption_income_debt_figure(bsimb, csimb, ysimb)
fig_02 = consumption_debt_fanchart(csimb, cons_meanb, cons_varb,
                                   bsimb, debt_meanb, debt_varb)
```

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, reflecting that this is a closed borrower-and-lender economy
- Now the cointegrating residual seems stationary, and not just asymptotically stationary

Nonfinancial Income, Consumption, and Debt
y and c
t
c
y
debt
t
b

Let's have a look at the cointegration figure

```
fig = cointegration_figure(bsimb, csimb)
fig.show()
```

Search with Offer Distribution Unknown

Contents

- *Search with Offer Distribution Unknown*
  - *Overview*
  - *Model*
  - *Take 1: Solution by VFI*
  - *Take 2: A More Efficient Method*
  - *Exercises*

– *Solutions*

Overview

In this lecture we consider an extension of the job search model developed by John J. 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

- Based on the presentation in *[LS12]*, section 6.6

Model features

- Infinite horizon dynamic programming with two states and one binary control
- Bayesian updating to learn the unknown distribution

Model

Let's first recall the basic McCall model *[McC70]* and then add the variation we want to consider

The Basic McCall Model

Consider an unemployed worker who is presented in each period with a permanent 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 iid and generated from known density h

The worker aims to maximize the expected discounted sum of earnings $\mathbb{E}\sum_{t=0}^{\infty}\beta^t y_t$

Trade-off:

- Waiting too long for a good offer is costly, since the future is discounted
- Accepting too early is costly, since better offers will arrive with probability one

Let $V(w)$ denote the maximal expected discounted sum of earnings that can be obtained by an unemployed worker who starts with wage offer w in hand

The function V satisfies the recursion

$$V(w) = \max\left\{\frac{w}{1-\beta}, c + \beta \int V(w')h(w')dw'\right\} \tag{5.83}$$

where the two terms on the r.h.s. are the respective payoffs from accepting and rejecting the current offer w

The optimal policy is a map from states into actions, and hence a binary function of w

Not surprisingly, it turns out to have the form $\mathbf{1}\{w \geq \bar{w}\}$, where

- $\bar{w}$ is a constant depending on (β, h, c) called the *reservation wage*
- $\mathbf{1}\{w \geq \bar{w}\}$ is an indicator function returning 1 if $w \geq \bar{w}$ and 0 otherwise
- 1 indicates "accept" and 0 indicates "reject"

For further details see *[LS12]*, section 6.3

Offer Distribution Unknown

Now let's extend the model by considering the variation presented in *[LS12]*, 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 π_0 on f being chosen

Update rule: worker's time t estimate of the distribution is $\pi_t f + (1-\pi_t)g$, where π_t updates via

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1-\pi_t)g(w_{t+1})} \tag{5.84}$$

This last expression follows from Bayes' rule, which tells us that

$$\mathbb{P}\{h = f \,|\, W = w\} = \frac{\mathbb{P}\{W = w \,|\, h = f\}\mathbb{P}\{h = f\}}{\mathbb{P}\{W = w\}} \quad \text{and} \quad \mathbb{P}\{W = w\} = \sum_{\psi \in \{f,g\}} \mathbb{P}\{W = w \,|\, h = \psi\}\mathbb{P}\{h = \psi\}$$

The fact that (5.84) 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

$$V(w,\pi) = \max\left\{\frac{w}{1-\beta},\, c + \beta \int V(w',\pi')\, h_\pi(w')\, dw'\right\} \quad \text{where} \quad \pi' = q(w',\pi) \tag{5.85}$$

Notice that the current guess π is a state variable, since it affects the worker's perception of probabilities for future rewards

Parameterization

Following section 6.6 of *[LS12]*, our baseline parameterization will be

- f is $\mathrm{Beta}(1,1)$ scaled (i.e., draws are multiplied by) some factor w_m
- g is $\mathrm{Beta}(3,1.2)$ 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

Looking Forward

What kind of optimal policy might result from (5.85) and the parameterization specified above?

Intuitively, if we accept at w_a and $w_a \leq w_b$, then — all other things being given — we should also accept at w_b

This suggests a policy of accepting whenever w exceeds some threshold value $\bar{w}$

But $\bar{w}$ should depend on π — in fact it should be decreasing in π because

- f is a less attractive offer distribution than g
- larger π means more weight on f and less on g

Thus larger π depresses the worker's assessment of her future prospects, and relatively low current offers become more attractive

Summary: We conjecture that the optimal policy is of the form $\mathbb{1}\{w \geq \bar{w}(\pi)\}$ for some decreasing function $\bar{w}$

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

VFI is implemented in the file `odu/odu.py`, and can be downloaded here

The code is as follows

```
"""
Filename: odu.py

Authors: Thomas Sargent, John Stachurski

Solves the "Offer Distribution Unknown" Model by value function
iteration and a second faster method discussed in the corresponding
quantecon lecture.

"""
from scipy.interpolate import LinearNDInterpolator
from scipy.integrate import fixed_quad
from scipy.stats import beta as beta_distribution
from numpy import maximum as npmax
import numpy as np


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_max : scalar(float), optional(default=2)
        Maximum wage possible
    w_grid_size : scalar(int), optional(default=40)
        Size of the grid on wages
    pi_grid_size : scalar(int), optional(default=40)
        Size of the grid on probabilities

    Attributes
    ----------
    beta, c, w_max : see Parameters
    w_grid : np.ndarray
        Grid points over wages, ndim=1
    pi_grid : np.ndarray
        Grid points over pi, ndim=1
    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_max
    G : scipy.stats._distn_infrastructure.rv_frozen
        Beta distribution with params (G_a, G_b), scaled by w_max
    f : function
        Density of F
    g : function
        Density of G
    pi_min : scalar(float)
        Minimum of grid over pi
    pi_max : scalar(float)
        Maximum of grid over pi
    """

    def __init__(self, beta=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, pi_grid_size=40):

        self.beta, self.c, self.w_max = beta, c, w_max
        self.F = beta_distribution(F_a, F_b, scale=w_max)
        self.G = beta_distribution(G_a, G_b, scale=w_max)
        self.f, self.g = self.F.pdf, self.G.pdf    # Density functions
        self.pi_min, self.pi_max = 1e-3, 1 - 1e-3  # Avoids instability
        self.w_grid = np.linspace(0, w_max, w_grid_size)
        self.pi_grid = np.linspace(self.pi_min, self.pi_max, pi_grid_size)
        x, y = np.meshgrid(self.w_grid, self.pi_grid)
        self.grid_points = np.column_stack((x.ravel(1), y.ravel(1)))


    def q(self, w, pi):
        """
```

```
        Updates pi using Bayes' rule and the current wage observation w.

        Returns
        -------

        new_pi : scalar(float)
            The updated probability

        """

        new_pi = 1.0 / (1 + ((1 - pi) * self.g(w)) / (pi * self.f(w)))

        # Return new_pi when in [pi_min, pi_max] and else end points
        new_pi = np.maximum(np.minimum(new_pi, self.pi_max), self.pi_min)

        return new_pi

    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(pi_grid))
            An approximate value function represented as a
            one-dimensional array.

        Returns
        -------
        new_v : array_like(float, ndim=1, length=len(pi_grid))
            The updated value function

        """
        # == Simplify names == #
        f, g, beta, c, q = self.f, self.g, self.beta, self.c, self.q

        vf = LinearNDInterpolator(self.grid_points, v)
        N = len(v)
        new_v = np.empty(N)

        for i in range(N):
            w, pi = self.grid_points[i, :]
            v1 = w / (1 - beta)
            integrand = lambda m: vf(m, q(m, pi)) * (pi * f(m)
                                                     + (1 - pi) * g(m))
            integral, error = fixed_quad(integrand, 0, self.w_max)
            v2 = c + beta * 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(pi_grid))
            An approximate value function represented as a
            one-dimensional array.

        Returns
        -------
        policy : array_like(float, ndim=1, length=len(pi_grid))
            The decision to accept or reject an offer where 1 indicates
            accept and 0 indicates reject

        """
        # == Simplify names == #
        f, g, beta, c, q = self.f, self.g, self.beta, 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, pi = self.grid_points[i, :]
            v1 = w / (1 - beta)
            integrand = lambda m: vf(m, q(m, pi)) * (pi * f(m) +
                                                     (1 - pi) * g(m))
            integral, error = fixed_quad(integrand, 0, self.w_max)
            v2 = c + beta * integral
            policy[i] = v1 > v2  # Evaluates to 1 or 0

        return policy

    def res_wage_operator(self, phi):
        """

        Updates the reservation wage function guess phi via the operator
        Q.

        Parameters
        ----------
        phi : array_like(float, ndim=1, length=len(pi_grid))
            This is reservation wage guess

        Returns
        -------
        new_phi : array_like(float, ndim=1, length=len(pi_grid))
            The updated reservation wage guess.

        """
```

```
        # == Simplify names == #
        beta, c, f, g, q = self.beta, self.c, self.f, self.g, self.q
        # == Turn phi into a function == #
        phi_f = lambda p: np.interp(p, self.pi_grid, phi)

        new_phi = np.empty(len(phi))
        for i, pi in enumerate(self.pi_grid):
            def integrand(x):
                "Integral expression on right-hand side of operator"
                return npmax(x, phi_f(q(x, pi))) * (pi*f(x) + (1 - pi)*g(x))
            integral, error = fixed_quad(integrand, 0, self.w_max)
            new_phi[i] = (1 - beta) * c + beta * integral

        return new_phi
```

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 let's look quickly at solutions computed from value function iteration

Here's the value function:

The optimal policy:

Code for producing these figures can be found here

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
- decreasing as expected

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

Uses iteration with an operator having 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 $w = \bar{w}(\pi)$, the worker is indifferent between accepting and rejecting

Hence the two choices on the right-hand side of (5.85) have equal value:

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int V(w', \pi')\, h_\pi(w')\, dw' \tag{5.86}$$

Together, (5.85) and (5.86) give

$$V(w, \pi) = \max\left\{\frac{w}{1-\beta}, \frac{\bar{w}(\pi)}{1-\beta}\right\} \tag{5.87}$$

Combining (5.86) and (5.87), we obtain

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int \max\left\{\frac{w'}{1-\beta}, \frac{\bar{w}(\pi')}{1-\beta}\right\} h_\pi(w')\, dw'$$

Multiplying by $1-\beta$, substituting in $\pi' = q(w', \pi)$ and using $\circ$ for composition of functions yields

$$\bar{w}(\pi) = (1-\beta)c + \beta \int \max\left\{w', \bar{w} \circ q(w', \pi)\right\} h_\pi(w')\, dw' \tag{5.88}$$

Equation (5.88) can be understood as a functional equation, where $\bar{w}$ is the unknown function

- Let's call it the *reservation wage functional equation* (RWFE)
- The solution $\bar{w}$ to the RWFE is the object that we wish to compute

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

$$(Q\psi)(\pi) = (1-\beta)c + \beta \int \max\left\{w', \psi \circ q(w', \pi)\right\} h_\pi(w')\, dw' \tag{5.89}$$

Comparing (5.88) and (5.89), 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 (5.88) and vice versa

Moreover, for any $\psi, \phi \in b[0,1]$, basic algebra and the triangle inequality for integrals tells us that

$$|(Q\psi)(\pi) - (Q\phi)(\pi)| \leq \beta \int \left|\max\left\{w', \psi \circ q(w', \pi)\right\} - \max\left\{w', \phi \circ q(w', \pi)\right\}\right| h_\pi(w')\, dw' \tag{5.90}$$

Working case by case, it is easy to check that for real numbers a, b, c we always have

$$|\max\{a, b\} - \max\{a, c\}| \leq |b - c| \tag{5.91}$$

Combining (5.90) and (5.91) yields

$$|(Q\psi)(\pi) - (Q\phi)(\pi)| \leq \beta \int \left|\psi \circ q(w', \pi) - \phi \circ q(w', \pi)\right| \, h_\pi(w') \, dw' \leq \beta \|\psi - \phi\| \tag{5.92}$$

Taking the supremum over π now gives us

$$\|Q\psi - Q\phi\| \leq \beta \|\psi - \phi\| \tag{5.93}$$

In other words, Q is a contraction of modulus β on the complete metric space $(b[0, 1], \|\cdot\|)$

Hence

- A unique solution $\bar{w}$ to the RWFE exists in $b[0, 1]$
- $Q^k \psi \to \bar{w}$ 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 $\bar{w}$

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

Solutions

```
import numpy as np
import matplotlib.pyplot as plt
from quantecon import compute_fixed_point
```

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`

```
sp = SearchProblem(pi_grid_size=50)

phi_init = np.ones(len(sp.pi_grid))
w_bar = compute_fixed_point(sp.res_wage_operator, phi_init)

fig, ax = plt.subplots(figsize=(9, 7))
ax.plot(sp.pi_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.pi_grid, 0, w_bar, color='blue', alpha=0.15)
ax.fill_between(sp.pi_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()
```

```
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
5            2.829e-02      1.071e-01
10           5.174e-03      2.177e-01
15           9.652e-04      3.220e-01
Converged in 15 steps
```

The next piece of code is not one of the exercises from quant-econ, it's 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

```
# 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
phi_init = np.ones(len(sp.pi_grid))
w_bar_vals = compute_fixed_point(sp.res_wage_operator, phi_init)
w_bar = lambda x: np.interp(x, pi_grid, w_bar_vals)


class Agent:
    """
    Holds the employment state and beliefs of an individual agent.
    """

    def __init__(self, pi=1e-3):
```

```
        self.pi = pi
        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.pi):
                self.employed = 1
            else:
                self.pi = 1.0 / (1 + ((1 - self.pi) * g(w)) / (self.pi *␣
↪f(w)))


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("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)
    employed = [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()
```

```
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
5            2.829e-02      1.090e-01
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
```

Robustness

Contents

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

Overview

This lecture modifies a Bellman equation to express a decision maker's doubts about transition dynamics

His specification doubts make the decision maker want a *robust* decision rule

Robust means insensitive to misspecification of transition dynamics

The decision maker has a single *approximating model*

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

He fears that outcomes will actually be determined by another model that he cannot describe explicitly

All that he knows is that the actual data-generating model is in some (uncountable) set of models that surrounds his approximating model

He quantifies the discrepancy between his approximating model and the genuine data-generating model by using a quantity called *entropy*

(We'll explain what entropy means below)

He wants a decision rule that will work well enough no matter which of those other models actually governs outcomes

This is what it means for his decision rule to be "robust to misspecification of an approximating model"

This may sound like too much to ask for, but . . .

. . . a *secret weapon* is available to design robust decision rules

The secret weapon is max-min control theory

A value-maximizing decision maker enlists the aid of an (imaginary) value-minimizing model chooser to construct *bounds* on the value attained by a given decision rule under different models of the transition dynamics

The original decision maker uses those bounds to construct a decision rule with an assured performance level, no matter which model actually governs outcomes

---

Note: In reading this lecture, please don't think that our decision maker is paranoid when he conducts a worst-case analysis. By designing a rule that works well against a worst-case, his intention is to construct a rule that will work well across a *set* of models.

Sets of Models Imply Sets Of Values

Our "robust" decision maker wants to know how well a given rule will work when he does not *know* a single transition law . . .

. . . he wants to know *sets* of values that will be attained by a given decision rule F under a *set* of transition laws

Ultimately, he wants to design a decision rule F that shapes these *sets* of values in ways that he prefers

With this in mind, consider the following graph, which relates to a particular decision problem to be explained below

The figure shows a *value-entropy correspondence* for a particular decision rule F

The shaded set is the graph of the correspondence, which maps entropy to a set of values associated with a set of models that surround the decision maker's approximating model

Here

- *Value* refers to a sum of discounted rewards obtained by applying the decision rule F when the state starts at some fixed initial state x_0
- *Entropy* is a nonnegative number that measures the size of a set of models surrounding the decision maker's approximating model
  - Entropy is zero when the set includes only the approximating model, indicating that the decision maker completely trusts the approximating model
  - Entropy is bigger, and the set of surrounding models is bigger, the less the decision maker trusts the approximating model

The shaded region indicates that for **all** models having entropy less than or equal to the number on the horizontal axis, the value obtained will be somewhere within the indicated set of values

Now let's compare sets of values associated with two different decision rules, F_r and F_b

In the next figure,

- The red set shows the value-entropy correspondence for decision rule F_r
- The blue set shows the value-entropy correspondence for decision rule F_b

The blue correspondence is skinnier than the red correspondence

This conveys the sense in which the decision rule F_b is *more robust* than the decision rule F_r

- *more robust* means that the set of values is less sensitive to *increasing misspecification* as measured by entropy

Notice that the less robust rule F_r promises higher values for small misspecifications (small entropy)

(But it is more fragile in the sense that it is more sensitive to perturbations of the approximating model)

Below we'll explain in detail how to construct these sets of values for a given F, but for now . . .

Here is a hint about the *secret weapons* we'll use to construct these sets

- We'll use some min problems to construct the lower bounds
- We'll use some max problems to construct the upper bounds

We will also describe how to choose F to shape the sets of values

This will involve crafting a *skinnier* set at the cost of a lower *level* (at least for low values of entropy)

Inspiring Video

If you want to understand more about why one serious quantitative researcher is interested in this approach, we recommend Lars Peter Hansen's Nobel lecture

Other References

Our discussion in this lecture is based on

- *[HS00]*
- *[HS08]*

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

$$\sum_{t=0}^{\infty} \beta^t \left\{ x_t' R x_t + u_t' Q u_t \right\} \tag{5.94}$$

subject to the linear law of motion

$$x_{t+1} = A x_t + B u_t + C w_{t+1}, \qquad t = 0, 1, 2, \ldots \tag{5.95}$$

As before,

- x_t is $n \times 1$, A is $n \times n$
- u_t is $k \times 1$, B is $n \times k$
- w_t is $j \times 1$, C is $n \times j$
- R is $n \times n$ and Q is $k \times k$

Here x_t is the state, u_t is the control, and w_t is a shock vector.

For now we take $\{w_t\} := \{w_t\}_{t=1}^{\infty}$ to be deterministic — a single fixed sequence

We also allow for *model uncertainty* on the part of the agent solving this optimization problem

In particular, the agent takes $w_t = 0$ for all $t \geq 0$ as a benchmark model, but admits the possibility that this model might be wrong

As a consequence, she also considers a set of alternative models expressed in terms of sequences $\{w_t\}$ that are "close" to the zero sequence

She seeks a policy that will do well enough for a set of alternative models whose members are pinned down by sequences $\{w_t\}$

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

Constructing More Robust Policies

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

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

(Here J depends on t because the sequence $\{w_t\}$ is not recursive)

Our tool for studying robustness is to construct a rule that works well even if an adverse sequence $\{w_t\}$ occurs

In our framework, "adverse" means "loss increasing"

As we'll see, this will eventually lead us to construct the Bellman equation

$$J(x) = \min_u \max_w \{x'Rx + u'Qu + \beta\, [J(Ax + Bu + Cw) - \theta w'w]\} \tag{5.96}$$

Notice that we've added the penalty term $-\theta w'w$

Since $w'w = \|w\|^2$, this term becomes influential when w moves away from the origin

The penalty parameter θ controls how much we penalize the maximizing agent for "harming" the minmizing agent

By raising θ more and more, we more and more limit the ability of maximizing agent to distort outcomes relative to the approximating model

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

Analyzing the Bellman equation

So what does J in (5.96) look like?

As with the *ordinary LQ control model*, J takes the form $J(x) = x'Px$ for some symmetric positive definite matrix P

One of our main tasks will be to analyze and compute the matrix P

Related tasks will be to study associated feedback rules for u_t and w_{t+1}

First, using *matrix calculus*, you will be able to verify that

$$\begin{aligned} \max_w \{(Ax + Bu + Cw)'P(Ax + Bu + Cw) - \theta w'w\} \\ = (Ax + Bu)'\mathcal{D}(P)(Ax + Bu) \end{aligned} \tag{5.97}$$

where

$$\mathcal{D}(P) := P + PC(\theta I - C'PC)^{-1}C'P \tag{5.98}$$

and I is a $j \times j$ identity matrix. Substituting this expression for the maximum into (5.96) yields

$$x'Px = \min_u \{x'Rx + u'Qu + \beta\,(Ax + Bu)'\mathcal{D}(P)(Ax + Bu)\} \tag{5.99}$$

Using similar mathematics, the solution to this minimization problem is $u = -Fx$ where $F := (Q + \beta B'\mathcal{D}(P)B)^{-1}\beta B'\mathcal{D}(P)A$

Substituting this minimizer back into (5.99) and working through the algebra gives $x'Px = x'\mathcal{B}(\mathcal{D}(P))x$ for all x, or, equivalently,

$$P = \mathcal{B}(\mathcal{D}(P))$$

where $\mathcal{D}$ is the operator defined in (5.98) and

$$\mathcal{B}(P) := R - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA + \beta A'PA$$

The operator $\mathcal{B}$ is the standard (i.e., non-robust) LQ Bellman operator, and $P = \mathcal{B}(P)$ is the standard matrix Riccati equation coming from the Bellman equation — see *this discussion*

Under some regularity conditions (see *[HS08]*), the operator $\mathcal{B} \circ \mathcal{D}$ has a unique positive definite fixed point, which we denote below by $\hat{P}$

A robust policy, indexed by θ, is $u = -\hat{F}x$ where

$$\hat{F} := (Q + \beta B'\mathcal{D}(\hat{P})B)^{-1}\beta B'\mathcal{D}(\hat{P})A \tag{5.100}$$

We also define

$$\hat{K} := (\theta I - C'\hat{P}C)^{-1}C'\hat{P}(A - B\hat{F}) \tag{5.101}$$

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 (5.97) evaluated at the fixed rule $u = -\hat{F}x$

Note that $\hat{P}, \hat{F}, \hat{K}$ are all determined by the primitives and θ

Note also that if θ is very large, then $\mathcal{D}$ is approximately equal to the identity mapping

Hence, when θ is large, $\hat{P}$ and $\hat{F}$ are approximately equal to their standard LQ values

Furthermore, when θ is large, $\hat{K}$ is approximately equal to zero

Conversely, smaller θ is associated with greater fear of model misspecification, and greater concern for robustness

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

Agent 2

1. knows a fixed policy F specifying the behavior of agent 1, in the sense that $u_t = -Fx_t$ for all t
2. responds by choosing a shock sequence $\{w_t\}$ from a set of paths sufficiently close to the benchmark sequence $\{0, 0, 0, \ldots\}$

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 recusive formulation, it turns out to be convenient to restrict a discounted inner product

$$\sum_{t=1}^{\infty} \beta^t w_t' w_t \leq \eta \tag{5.102}$$

Now let F be a fixed policy, and let $J_F(x_0, \mathbf{w})$ be the present-value cost of that policy given sequence $\mathbf{w} := \{w_t\}$ and initial condition $x_0 \in \mathbb{R}^n$

Substituting $-Fx_t$ for u_t in (5.94), this value can be written as

$$J_F(x_0, \mathbf{w}) := \sum_{t=0}^{\infty} \beta^t x_t' (R + F'QF) x_t \tag{5.103}$$

where

$$x_{t+1} = (A - BF)x_t + Cw_{t+1} \tag{5.104}$$

and the initial condition x_0 is as specified in the left side of (5.103)

Agent 2 chooses $\mathbf{w}$ to maximize agent 1's loss $J_F(x_0, \mathbf{w})$ subject to (5.102)

Using a Lagrangian formulation, we can express this problem as

$$\max_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \left\{ x_t'(R + F'QF)x_t - \beta\theta(w_{t+1}' w_{t+1} - \eta) \right\}$$

where $\{x_t\}$ satisfied (5.104) and θ is a Lagrange multiplier on constraint (5.102)

For the moment, let's take θ as fixed, allowing us to drop the constant $\beta\theta\eta$ term in the objective function, and hence write the problem as

$$\max_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \left\{ x_t'(R + F'QF)x_t - \beta\theta w_{t+1}' w_{t+1} \right\}$$

or, equivalently,

$$\min_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t + \beta\theta w_{t+1}' w_{t+1} \right\} \tag{5.105}$$

subject to (5.104)

What's striking about this optimization problem is that it is once again an LQ discounted dynamic programming problem, with $\mathbf{w} = \{w_t\}$ as the sequence of controls

The expression for the optimal policy can be found by applying the usual LQ formula (*see here*)

We denote it by $K(F, \theta)$, with the interpretation $w_{t+1} = K(F, \theta)x_t$

The remaining step for agent 2's problem is to set θ to enforce the constraint (5.102), which can be done by choosing $\theta = \theta_\eta$ such that

$$\beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \theta_\eta)' K(F, \theta_\eta) x_t = \eta \tag{5.106}$$

Here x_t is given by (5.104) — 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 (5.105) as $R_\theta(x_0, F)$.

Because "minimizers minimize" we have

$$R_\theta(x_0, F) \leq \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t \right\} + \beta\theta \sum_{t=0}^{\infty} \beta^t w_{t+1}' w_{t+1},$$

where $x_{t+1} = (A - BF + CK(F, \theta))x_t$ and x_0 is a given initial condition.

This inequality in turn implies the inequality

$$R_\theta(x_0, F) - \theta \text{ ent} \leq \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t \right\} \tag{5.107}$$

where

$$\text{ent} := \beta \sum_{t=0}^{\infty} \beta^t w_{t+1}' w_{t+1}$$

The left side of inequality (5.107) 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 (5.107) is attained when

$$\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x_t' K(F,\theta)' K(F,\theta) x_t \tag{5.108}$$

To construct the *lower bound* on the set of values associated with all perturbations $\mathbf{w}$ satisfying the entropy constraint (5.102) at a given entropy level, we proceed as follows:

- For a given θ, solve the minimization problem (5.105)
- Compute the minimizer $R_\theta(x_0, F)$ and the associated entropy using (5.108)
- Compute the lower bound on the value function $R_\theta(x_0, F) - \theta \text{ ent}$ and plot it against ent
- Repeat the preceding three steps for a range of values of θ to trace out the lower bound

---

Note: This procedure sweeps out a set of separating hyperplanes indexed by different values for the Lagrange multiplier θ

The Upper Bound

To construct an *upper bound* we use a very similar procedure

We simply replace the *minimization* problem (5.105) with the *maximization* problem

$$V_{\tilde\theta}(x_0, F) = \max_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t - \beta\tilde\theta w_{t+1}' w_{t+1} \right\} \tag{5.109}$$

where now $\tilde\theta > 0$ penalizes the choice of $\mathbf{w}$ with larger entropy.

(Notice that $\tilde\theta = -\theta$ in problem (5.105))

Because "maximizers maximize" we have

$$V_{\tilde\theta}(x_0, F) \geq \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t \right\} - \beta\tilde\theta \sum_{t=0}^{\infty} \beta^t w_{t+1}' w_{t+1}$$

which in turn implies the inequality

$$V_{\tilde\theta}(x_0, F) + \tilde\theta \text{ ent} \geq \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t \right\} \tag{5.110}$$

where

$$\text{ent} \equiv \beta \sum_{t=0}^{\infty} \beta^t w_{t+1}' w_{t+1}$$

The left side of inequality (5.110) is a straight line with slope $\tilde\theta$

The upper bound on the left side of (5.110) is attained when

$$\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \tilde{\theta})' K(F, \tilde{\theta}) x_t \tag{5.111}$$

To construct the *upper bound* on the set of values associated all perturbations $\mathbf{w}$ with a given entropy we proceed much as we did for the lower bound

- For a given $\tilde{\theta}$, solve the maximization problem (5.109)
- Compute the maximizer $V_{\tilde{\theta}}(x_0, F)$ and the associated entropy using (5.111)
- Compute the upper bound on the value function $V_{\tilde{\theta}}(x_0, F) + \tilde{\theta}\ \text{ent}$ 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

$$\min_{\{u_t\}} \sum_{t=0}^{\infty} \beta^t \left\{ x_t' R x_t + u_t' Q u_t - \beta \theta w_{t+1}' w_{t+1} \right\} \tag{5.112}$$

where $\{w_{t+1}\}$ satisfies $w_{t+1} = K x_t$

In other words, agent 1 minimizes

$$\sum_{t=0}^{\infty} \beta^t \left\{ x_t' (R - \beta \theta K' K) x_t + u_t' Q u_t \right\} \tag{5.113}$$

subject to

$$x_{t+1} = (A + CK) x_t + B u_t \tag{5.114}$$

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 Φ, where

$$\tilde{F} = \Phi(K(F, \theta))$$

The map $F \mapsto \Phi(K(F, \theta))$ corresponds to a situation in which

1. agent 1 uses an arbitrary initial policy F
2. agent 2 best responds to agent 1 by choosing $K(F, \theta)$
3. agent 1 best responds to agent 2 by choosing $\tilde{F} = \Phi(K(F, \theta))$

As you may have already guessed, the robust policy $\hat{F}$ defined in (5.100) is a fixed point of the mapping Φ

In particular, for any given θ,

1. $K(\hat{F}, \theta) = \hat{K}$, where $\hat{K}$ is as given in (5.101)
2. $\Phi(\hat{K}) = \hat{F}$

A sketch of the proof is given in *the appendix*

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 (5.96) with the stochastic analogue

$$J(x) = \min_u \max_{\psi \in \mathcal{P}} \left\{ x'Rx + u'Qu + \beta \left[\int J(Ax + Bu + Cw) \, \psi(dw) - \theta D_{KL}(\psi, \phi) \right] \right\} \tag{5.115}$$

Here $\mathcal{P}$ represents the set of all densities on $\mathbb{R}^n$ and ϕ is the benchmark distribution $N(0, I)$

The distribution ϕ is chosen as the least desirable conditional distribution in terms of next period outcomes, while taking into account the penalty term $\theta D_{KL}(\psi, \phi)$

This penalty term plays a role analogous to the one played by the deterministic penalty $\theta w'w$ in (5.96), since it discourages large deviations from the benchmark

Solving the Model

The maximization problem in (5.115) appears highly nontrivial — after all, we are maximizing over an infinite dimensional space consisting of the entire set of densities

However, it turns out that the solution is tractable, and in fact also falls within the class of normal distributions

First, we note that J has the form $J(x) = x'Px + d$ for some positive definite matrix P and constant real number d

Moreover, it turns out that if $(I - \theta^{-1}C'PC)^{-1}$ is nonsingular, then

$$\begin{aligned}\max_{\psi \in \mathcal{P}} \left\{ \int (Ax + Bu + Cw)'P(Ax + Bu + Cw)\, \psi(dw) - \theta D_{KL}(\psi, \phi) \right\} \\ = (Ax + Bu)'\mathcal{D}(P)(Ax + Bu) + \kappa(\theta, P)\end{aligned} \tag{5.116}$$

where

$$\kappa(\theta, P) := \theta \ln[\det(I - \theta^{-1}C'PC)^{-1}]$$

and the maximizer is the Gaussian distribution

$$\psi = N\left((\theta I - C'PC)^{-1}C'P(Ax + Bu), (I - \theta^{-1}C'PC)^{-1}\right) \tag{5.117}$$

Substituting the expression for the maximum into Bellman equation (5.115) and using $J(x) = x'Px + d$ gives

$$x'Px + d = \min_u \left\{ x'Rx + u'Qu + \beta\,(Ax + Bu)'\mathcal{D}(P)(Ax + Bu) + \beta\,[d + \kappa(\theta, P)] \right\} \tag{5.118}$$

Since constant terms do not affect minimizers, the solution is the same as (5.99), leading to

$$x'Px + d = x'\mathcal{B}(\mathcal{D}(P))x + \beta\,[d + \kappa(\theta, P)]$$

To solve this Bellman equation, we take $\hat{P}$ to be the positive definite fixed point of $\mathcal{B} \circ \mathcal{D}$

In addition, we take $\hat{d}$ as the real number solving $d = \beta\,[d + \kappa(\theta, P)]$, which is

$$\hat{d} := \frac{\beta}{1 - \beta}\kappa(\theta, P) \tag{5.119}$$

The robust policy in this stochastic case is the minimizer in (5.118), which is once again $u = -\hat{F}x$ for $\hat{F}$ given by (5.100)

Substituting the robust policy into (5.117) 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 (5.101)

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 (5.115), satisfies

$$J_F(x) = \max_{\psi \in \mathcal{P}} \left\{ x'(R + F'QF)x + \beta \left[\int J_F((A - BF)x + Cw)\, \psi(dw) - \theta D_{KL}(\psi, \phi) \right] \right\}$$

Writing $J_F(x) = x'P_F x + d_F$ and applying the same argument used to derive (5.116) we get

$$x'P_F x + d_F = x'(R + F'QF)x + \beta \left[x'(A - BF)'\mathcal{D}(P_F)(A - BF)x + d_F + \kappa(\theta, P_F) \right]$$

To solve this we take P_F to be the fixed point

$$P_F = R + F'QF + \beta(A - BF)'\mathcal{D}(P_F)(A - BF)$$

and

$$d_F := \frac{\beta}{1-\beta}\kappa(\theta, P_F) = \frac{\beta}{1-\beta}\theta \ln[\det(I - \theta^{-1}C'P_F C)^{-1}] \tag{5.120}$$

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

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 (5.100) – (5.101) 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 (5.106)
- `evaluate_F()` computes the loss and entropy associated with a given policy — see *this discussion*

Application

Let us consider a monopolist similar to *this one*, but now facing model uncertainty

The inverse demand function is $p_t = a_0 - a_1 y_t + d_t$

where

$$d_{t+1} = \rho d_t + \sigma_d w_{t+1}, \quad \{w_t\} \overset{\text{iid}}{\sim} N(0,1)$$

and all parameters are strictly positive

The period return function for the monopolist is

$$r_t = p_t y_t - \gamma \frac{(y_{t+1} - y_t)^2}{2} - c y_t$$

Its objective is to maximize expected discounted profits, or, equivalently, to minimize $\mathbb{E} \sum_{t=0}^{\infty} \beta^t (-r_t)$

To form a linear regulator problem, we take the state and control to be

$$x_t = \begin{bmatrix} 1 \\ y_t \\ d_t \end{bmatrix} \quad \text{and} \quad u_t = y_{t+1} - y_t$$

Setting $b := (a_0 - c)/2$ we define

$$R = -\begin{bmatrix} 0 & b & 0 \\ b & -a_1 & 1/2 \\ 0 & 1/2 & 0 \end{bmatrix} \quad \text{and} \quad Q = \gamma/2$$

For the transition matrices we set

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \rho \end{bmatrix}, \qquad B = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \qquad C = \begin{bmatrix} 0 \\ 0 \\ \sigma_d \end{bmatrix}$$

Our aim is to compute the value-entropy correspondences *shown above*

The parameters are

$$a_0 = 100, a_1 = 0.5, \rho = 0.9, \sigma_d = 0.05, \beta = 0.95, c = 2, \gamma = 50.0$$

The standard normal distribution for w_t is understood as the agent's baseline, with uncertainty parameterized by θ

We compute value-entropy correspondences for two policies

1. The no concern for robustness policy F_0, which is the ordinary LQ loss minimizer
2. A "moderate" concern for robustness policy F_b, with $\theta = 0.02$

The code for producing the graph shown above, with blue being for the robust policy, is given in robust_monopolist.py

We repeat it here for convenience

```
"""
Filename: robust_monopolist.py
Authors: Chase Coleman, Spencer Lyon, Thomas Sargent, John Stachurski

The robust control problem for a monopolist with adjustment costs.  The
inverse demand curve is:

  p_t = a_0 - a_1 y_t + d_t

where d_{t+1} = \rho d_t + \sigma_d w_{t+1} for w_t ~ N(0, 1) and iid.
The period return function for the monopolist is

  r_t =  p_t y_t - gamma (y_{t+1} - y_t)^2 / 2 - c y_t

The objective of the firm is E_t \sum_{t=0}^\infty \beta^t r_t

For the linear regulator, we take the state and control to be

    x_t = (1, y_t, d_t) and u_t = y_{t+1} - y_t

"""
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
rho     = 0.9
sigma_d = 0.05
beta    = 0.95
c       = 2
gamma   = 50.0

theta = 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 = gamma / 2

A = np.array([[1., 0., 0.],
              [0., 1., 0.],
              [0., 0., rho]])
```

```
B = np.array([[0.],
              [1.],
              [0.]])
C = np.array([[0.],
              [0.],
              [sigma_d]])

# ------------------------------------------------------------------------ #
#                                  Functions
# ------------------------------------------------------------------------ #


def evaluate_policy(theta, F):
    """
    Given theta (scalar, dtype=float) and policy F (array_like), returns the
    value associated with that policy under the worst case path for {w_t}, as
    well as the entropy level.
    """
    rlq = qe.robustlq.RBLQ(Q, R, A, B, C, beta, theta)
    K_F, P_F, d_F, O_F, o_F = rlq.evaluate_F(F)
    x0 = np.array([[1.], [0.], [0.]])
    value = - x0.T.dot(P_F.dot(x0)) - d_F
    entropy = x0.T.dot(O_F.dot(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 theta 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':
```

```
        thetas = 1 / np.linspace(1e-8, 1000, grid_size)
    else:
        thetas = -1 / np.linspace(1e-8, 1000, grid_size)

    df = pd.DataFrame(index=thetas, columns=('value', 'entropy'))

    for theta in thetas:
        df.ix[theta] = evaluate_policy(theta, F)
        if df.ix[theta, 'entropy'] >= emax:
            break

    df = df.dropna(how='any')
    return df


# ------------------------------------------------------------------------- #
#                                     Main
# ------------------------------------------------------------------------- #


# == Compute the optimal rule == #
optimal_lq = qe.lqcontrol.LQ(Q, R, A, B, C, beta)
Po, Fo, do = optimal_lq.stationary_values()

# == Compute a robust rule given theta == #
baseline_robust = qe.robustlq.RBLQ(Q, R, A, B, C, beta, theta)
Fb, Kb, Pb = baseline_robust.robust_rule()

# == Check the positive definiteness of worst-case covariance matrix to == #
# == ensure that theta exceeds the breakdown point == #
test_matrix = np.identity(Pb.shape[0]) - np.dot(C.T, Pb.dot(C)) / theta
eigenvals, eigenvecs = eig(test_matrix)
assert (eigenvals >= 0).all(), 'theta below breakdown point.'


emax = 1.6e6

optimal_best_case = value_and_entropy(emax, Fo, 'best')
robust_best_case = value_and_entropy(emax, Fb, 'best')
optimal_worst_case = value_and_entropy(emax, Fo, 'worst')
robust_worst_case = value_and_entropy(emax, Fb, 'worst')

fig, ax = plt.subplots()

ax.set_xlim(0, emax)
ax.set_ylabel("Value")
ax.set_xlabel("Entropy")
ax.grid()

for axis in 'x', 'y':
    plt.ticklabel_format(style='sci', axis=axis, scilimits=(0, 0))

plot_args = {'lw': 2, 'alpha': 0.7}
```

```
colors = 'r', 'b'

df_pairs = ((optimal_best_case, optimal_worst_case),
            (robust_best_case, robust_worst_case))


class Curve:

    def __init__(self, x, y):
        self.x, self.y = x, y

    def __call__(self, z):
        return np.interp(z, self.x, self.y)


for c, df_pair in zip(colors, df_pairs):
    curves = []
    for df in df_pair:
        # == Plot curves == #
        x, y = df['entropy'], df['value']
        x, y = (np.asarray(a, dtype='float') for a in (x, y))
        egrid = np.linspace(0, emax, 100)
        curve = Curve(x, y)
        print(ax.plot(egrid, curve(egrid), color=c, **plot_args))
        curves.append(curve)
    # == Color fill between curves == #
    ax.fill_between(egrid,
                    curves[0](egrid),
                    curves[1](egrid),
                    color=c, alpha=0.1)

plt.show()
```

Here's another such figure, with $\theta = 0.002$ instead of 0.02

Can you explain the different shape of the value-entropy correspondence for the robust policy?

Appendix

We sketch the proof only of the first claim in *this section*, which is that, for any given θ, $K(\hat{F},\theta) = \hat{K}$, where $\hat{K}$ is as given in (5.101)

This is the content of the next lemma

Lemma. If $\hat{P}$ is the fixed point of the map $\mathcal{B} \circ \mathcal{D}$ and $\hat{F}$ is the robust policy as given in (5.100), then

$$K(\hat{F},\theta) = (\theta I - C'\hat{P}C)^{-1}C'\hat{P}(A - B\hat{F}) \tag{5.121}$$

Proof: As a first step, observe that when $F = \hat{F}$, the Bellman equation associated with the LQ problem (5.104) – (5.105) is

$$\tilde{P} = -R - \hat{F}'Q\hat{F} - \beta^2(A - B\hat{F})'\tilde{P}C(\beta\theta I + \beta C'\tilde{P}C)^{-1}C'\tilde{P}(A - B\hat{F}) + \beta(A - B\hat{F})'\tilde{P}(A - B\hat{F}) \tag{5.122}$$

(revisit *this discussion* if you don't know where (5.122) comes from) and the optimal policy is

$$w_{t+1} = -\beta(\beta\theta I + \beta C'\tilde{P}C)^{-1}C'\tilde{P}(A - B\hat{F})x_t$$

Suppose for a moment that $-\hat{P}$ solves the Bellman equation (5.122)

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 (5.121)

Hence it remains only to show that $-\hat{P}$ solves (5.122), or, in other words,

$$\hat{P} = R + \hat{F}'Q\hat{F} + \beta(A - B\hat{F})'\hat{P}C(\theta I - C'\hat{P}C)^{-1}C'\hat{P}(A - B\hat{F}) + \beta(A - B\hat{F})'\hat{P}(A - B\hat{F})$$

Using the definition of $\mathcal{D}$, we can rewrite the right-hand side more simply as

$$R + \hat{F}'Q\hat{F} + \beta(A - B\hat{F})'\mathcal{D}(\hat{P})(A - B\hat{F})$$

Although it involves a substantial amount of algebra, it can be shown that the latter is just $\hat{P}$

(Hint: Use the fact that $\hat{P} = \mathcal{B}(\mathcal{D}(\hat{P}))$)

CHAPTER

SIX

PART 6: MULTIPLE AGENT MODELS

These lectures look at important economic models that also illustrate common equilibrium concepts.

Schelling's Segregation Model

Contents

- *Schelling's Segregation Model*
  - *Outline*
  - *The Model*
  - *Results*
  - *Exercises*
  - *Solutions*

Outline

In 1969, Thomas C. Schelling developed a simple but striking model of racial segregation *[Sch69]*

His model studies the dynamics of racially mixed neighborhoods

Like much of Schelling's work, the model shows how local interactions can lead to surprising aggregate structure

In particular, it shows that relatively mild preference for neighbors of similar race can lead in aggregate to the collapse of mixed neighborhoods, and high levels of segregation

In recognition of this and other research, Schelling was awarded the 2005 Nobel Prize in Economic Sciences (joint with Robert Aumann)

In this lecture we (in fact you) will build and run a version of Schelling's model

The Model

We will cover a variation of Schelling's 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

Results

Let's have a look at the results we got when we coded and ran this model

As discussed above, agents are initially mixed randomly together

But after several cycles they become segregated into distinct regions

Cycle 1
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

Cycle 2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

In this instance, the program terminated after 4 cycles through the set of agents, indicating that all agents had reached a state of happiness

What is striking about the pictures is how rapidly racial integration breaks down

This is despite the fact that people in the model don't actually mind living mixed with the other type

Even with these preferences, the outcome is a high degree of segregation

Exercises

Exercise 1

Implement and run this simulation for yourself

Consider the following structure for your program

Agents can be modeled as *objects*

Here's an indication of how they might look

```
* Data:

    * type (green or orange)
    * location

* Methods:
```

```
    * Determine whether happy or not given locations of other agents

    * If not happy, move

        * find a new location where happy
```

And here's some pseudocode for the main loop

```
while agents are still moving
    for agent in agents
        give agent the opportunity to move
```

Use 250 agents of each type

Solutions

Exercise 1

Here's one solution that does the job we want. If you feel like a further exercise you can probably speed up some of the computations and then increase the number of agents.

```
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('Cycle {}'.format(cycle_num - 1))
    plt.show()

# == Main == #

num_of_type_0 = 250
num_of_type_1 = 250
num_neighbors = 10      # Number of agents regarded as neighbors
require_same_type = 5   # Want at least this many neighbors to be same type

# == Create a list of agents == #
agents = [Agent(0) for i in range(num_of_type_0)]
agents.extend(Agent(1) for i in range(num_of_type_1))


count = 1
# ==  Loop until none wishes to move == #
while 1:
    print('Entering loop ', count)
```

```
    plot_distribution(agents, count)
    count += 1
    no_one_moved = True
    for agent in agents:
        old_location = agent.location
        agent.update(agents)
        if agent.location != old_location:
            no_one_moved = False
    if no_one_moved:
        break

print('Converged, terminating.')
```

```
Entering loop  1
```

```
Entering loop  2
```

```
Entering loop  3
```

```
Entering loop  4
```

```
Converged, terminating.
```

A Lake Model of Employment and Unemployment

Contents

- *A Lake Model of Employment and Unemployment*
  - *Overview*

- *The Model*
- *Implementation*
- *Dynamics of an Individual Worker*
- *Endogenous Job Finding Rate*
- *Exercises*
- *Solutions*
- *Lake Model Solutions*

Overview

This lecture describes what has come to be called a *lake model*

The lake model is a basic tool for modeling unemployment

It allows us to analyze

- flows between unemployment and employment
- how these flows influence steady state employment and unemployment rates

It is a good model for interpreting monthly labor department reports on gross and net jobs created and jobs destroyed

The "lakes" in the model are the pools of employed and unemployed

The "flows" between the lakes are caused by

- firing and hiring
- entry and exit from the labor force

For the first part of this lecture, the parameters governing transitions into and out of unemployment and employment are exogenous

Later, we'll determine some of these transition rates endogenously using the *McCall search model*

The only knowledge required for this lecture is

- basic *linear algebra* and probability
- some familiarity with *Markov chains*

We'll 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

The Model

The economy is inhabited by a very large number of ex-ante identical workers

The workers live forever, spending their lives moving between unemployment and employment

Their rates of transition between employment and unemployment are governed by the following parameters:

- λ, the job finding rate for currently unemployed workers
- α, the dismissal rate for currently employed workers
- b, the entry rate into the labor force
- d, the exit rate from the labor force

The growth rate of the labor force evidently equals $g = b - d$

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 $X_t := \begin{pmatrix} E_t \\ U_t \end{pmatrix}$, the law of motion for X is

$$X_{t+1} = AX_t \quad \text{where} \quad A := \begin{pmatrix} (1-d)(1-\alpha) & (1-d)\lambda \\ (1-d)\alpha + b & (1-d)(1-\lambda) + b \end{pmatrix}$$

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

$$\begin{pmatrix} E_{t+1}/N_{t+1} \\ U_{t+1}/N_{t+1} \end{pmatrix} = \frac{1}{1+g} A \begin{pmatrix} E_t/N_t \\ U_t/N_t \end{pmatrix}$$

Letting

$$x_t := \begin{pmatrix} e_t \\ u_t \end{pmatrix} = \begin{pmatrix} E_t/N_t \\ U_t/N_t \end{pmatrix}$$

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

Implementation

Let's code up these equations

To do this we're going to use a class that we'll call *LakeModel*

This class will

1. store the primitives α, λ, b, d
2. compute and store the implied objects $g, A, \hat{A}$
3. provide methods to simulate dynamics of the stocks and rates

4. provide a method to compute the state state of the rate

To write an nice implementation, there's an issue we have to address

Derived data such as A depend on the primitives like α and λ

If a user alters these primitives, we would ideally like derived data to update automatically

(For example, if a user changes the value of b for a given instance of the class, we would like $g = b - d$ to update automatically)

To achieve this outcome, we're going to use descriptors and decorators such as *@property*

If you need to refresh your understanding of how these work, consult *this lecture*

Here's the code:

```
"""

Provides a class that simulates the dynamics of unemployment and employment in
the lake model.

"""

import numpy as np

class LakeModel:
    r"""
    Solves the lake model and computes dynamics of unemployment stocks and
    rates.

    Parameters:
    ------------
    lmda: 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, lmda=0.283, alpha=0.013, b=0.0124, d=0.00822):
        self._lmda = lmda
        self._alpha = alpha
        self._b = b
        self._d = d

        self.compute_derived_values()

    def compute_derived_values(self):
        # Unpack names to simplify expression
        lmda, alpha, b, d = self._lmda, self._alpha, self._b, self._d

        self._g = b - d
```

```
        self._A = np.array([ [(1-d) * (1-alpha), (1-d) * lmda],
                             [(1-d) * alpha + b, (1-lmda) * (1-d) + b]])

        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 lmda(self):
        return self._lmda

    @lmda.setter
    def lmda(self, new_value):
        self._lmda = new_value
        self.compute_derived_values()

    @property
    def alpha(self):
        return self._alpha

    @alpha.setter
    def alpha(self, new_value):
        self._alpha = 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):
        r"""
        Finds the steady state of the system :math:`x_{t+1} = \hat A x_{t}`

        Returns
        --------
        xbar : steady state vector of employment and unemployment rates
        """
        x = 0.5 * np.ones(2)
        error = tol + 1
        while error > tol:
            new_x = self.A_hat @ x
            error = np.max(np.abs(new_x - x))
            x = new_x
        return x

    def simulate_stock_path(self, X0, T):
        r"""
        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):
        r"""
        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 α, derived objects like A will also change

```
lm = LakeModel()
lm.alpha
```

```
0.013
```

```
lm.A
```

```
array([[ 0.97888686,  0.28067374],
       [ 0.02529314,  0.72350626]])
```

```
lm.alpha = 2
lm.A
```

```
array([[-0.99178   ,  0.28067374],
       [ 1.99596   ,  0.72350626]])
```

Aggregate Dynamics

Let's run a simulation under the default parameters (see above) starting from $X_0 = (138, 12)$

```
import matplotlib.pyplot as plt
import matplotlib
matplotlib.style.use('ggplot')

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

E_0 = e_0 * N_0
U_0 = u_0 * N_0

fig, axes = plt.subplots(3, 1, figsize=(10, 8))
X_0 = (E_0, U_0)
X_path = np.vstack(lm.simulate_stock_path(X_0, T))

ax = axes[0]
```

```
ax.plot(X_path[:,0], '-b', lw=2, alpha=0.7)
ax.set_title(r'Employment')

ax = axes[1]
ax.plot(X_path[:,1], '-b', lw=2, alpha=0.7)
ax.set_title(r'Unemployment')

ax = axes[2]
ax.plot(X_path.sum(1), '-b', lw=2, alpha=0.7)
ax.set_title(r'Labor Force')

plt.tight_layout()
plt.show()
```

The aggregates E_t and U_t don't converge because their sum $E_t + U_t$ grows at rate g

On the other hand, the vector of employment and unemployment rates x_t can be in a steady state $\bar{x}$ if there exists an $\bar{x}$ such that

- $\bar{x} = \hat{A}\bar{x}$

- the components satisfy $\bar{e} + \bar{u} = 1$

This equation tells us that a steady state level $\bar{x}$ is an eigenvector of $\hat{A}$ associated with a unit eigenvalue

We also have $x_t \to \bar{x}$ as $t \to \infty$ provided that the remaining eigenvalue of $\hat{A}$ has modulus less that 1

This is the case for our default parameters:

```
lm = LakeModel()
e, f = np.linalg.eigvals(lm.A_hat)
abs(e), abs(f)
```

```
(1.0, 0.69530673783584618)
```

Let's look at the convergence of the unemployment and employment rate to steady state levels (dashed red line)

```
lm = LakeModel()
e_0 = 0.92     # Initial employment rate
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 = (e_0, u_0)
x_path = np.vstack(lm.simulate_rate_path(x_0, T))

ax = axes[0]
ax.plot(x_path[:,0], '-b', lw=2, alpha=0.5)
ax.hlines(xbar[0], 0, T, 'r', '--')
ax.set_title(r'Employment rate')

ax = axes[1]
ax.plot(x_path[:,1], '-b', lw=2, alpha=0.5)
ax.hlines(xbar[1], 0, T, 'r', '--')
ax.set_title(r'Unemployment rate')

plt.tight_layout()
plt.show()
```

Dynamics of an Individual Worker

A individual worker's employment dynamics are governed by a *finite state Markov process*

The worker can be in one of two states:

- $s_t = 0$ means unemployed
- $s_t = 1$ means employed

Let's start off under the assumption that $b = d = 0$

The associated transition matrix is then

$$P = \begin{pmatrix} 1 - \lambda & \lambda \\ \alpha & 1 - \alpha \end{pmatrix}$$

Let ψ_t denote the *marginal distribution* over employment / unemployment states for the worker at time t

As usual, we regard it as a row vector

We know *from an earlier discussion* that ψ_t follows the law of motion

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

We also know from the *lecture on finite Markov chains* that if $\alpha \in (0, 1)$ and $\lambda \in (0, 1)$, then P is stationary and ergodic, with a unique stationary distribution ψ^*

The unique stationary distribution is defined by

$$\psi^*[0] = \frac{\alpha}{\alpha + \lambda}$$

Not surprisingly, probability mass on the unemployment state increases with the dismissal rate and falls with the job finding rate 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

As stated above, 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.py*'s *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
```

```
alpha, lmda = lm.alpha, lm.lmda

P = [[1 - lmda, lmda],
     [alpha, 1 - alpha]]

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

ax = axes[0]
ax.plot(s_bar_u, '-b', lw=2, alpha=0.5)
ax.hlines(xbar[1], 0, T, 'r', '--')
ax.set_title(r'Percent of time unemployed')

ax = axes[1]
ax.plot(s_bar_e, '-b', lw=2, alpha=0.5)
ax.hlines(xbar[0], 0, T, 'r', '--')
ax.set_title(r'Percent of time employed')

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

Endogenous Job Finding Rate

We now make the hiring rate endogenous

The transition rate from unemployment to employment will be determined by the McCall search model *[McC70]*

All details relevant to the following discussion can be found in *our treatment* of that model

Reservation Wage

The most important thing to remember about the model is that optimal decisions are characterized by a reservation wage $\bar{w}$

- if the wage offer w in hand is greater than or equal to $\bar{w}$, then the worker accepts
- otherwise, the worker rejects

As we saw in *our discussion of the model*, the reservation wage depends on the wage offer distribution and the parameters

- α (the separation rate)
- β (the discount factor)
- γ (the offer arrival rate)
- c (unemployment compensation)

Linking the McCall Search Model to the Lake Model

Suppose that all workers inside a Lake Model behave according to the McCall search model

The exogenous probability of leaving employment remains α

But their optimal decision rules determine the probability λ of leaving unemployment

This is now

$$\lambda = \gamma \mathbb{P}\{w_t \geq \bar{w}\} = \gamma \sum_{w' \geq \bar{w}} p(w') \tag{6.1}$$

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 τ sufficient to finance total unemployment payments

To attain a balanced budget at a steady state, taxes, the steady state unemployment rate u, and the unemployment compensation rate must satisfy

$$\tau = uc$$

The lump sum tax applies to everyone, including unemployed workers

Thus, the post-tax income of an employed worker with wage w is $w - \tau$

The post-tax income of an unemployed worker is $c - \tau$.

For each specification (c, τ) of government policy, we can solve for the worker's optimal reservation wage

This determines λ via (6.1) evaluated at post tax wages, which in turn determines a steady state unemployment rate $u(c, \tau)$

For a given level of unemployment benefit c, we can solve for a tax that balances the budget in the steady state

$$\tau = u(c, \tau)c$$

To evaluate alternative government tax-unemployment compensation pairs, we require a welfare criterion

We use a steady state welfare criterion

$$W := e\,\mathbb{E}[V \,|\, \text{employed}] + u\,U$$

where the notation V and U is as defined in the *McCall search model lecture*

The wage offer distribution will be a discretized version of the distribution $\mathcal{N}(\log(20), 1)$, as shown in the next figure

We take a period to be a month

We set b and d to match monthly birth and death rates, respectively, in the U.S. population

- $b = 0.0124$
- $d = 0.00822$

Following *[DFH06]*, we set α, the hazard rate of leaving employment, to

- $\alpha = 0.013$

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 implements value function iteration

```
"""
Implements iteration on the Bellman equations to solve the McCall growth model

"""
```

```
import numpy as np
from quantecon.distributions import BetaBinomial
from numba import jit

# A default utility function

@jit
def u(c, sigma):
    if c > 0:
        return (c**(1 - sigma) - 1) / (1 - sigma)
    else:
        return -10e6


class McCallModel:
    """
    Stores the parameters and functions associated with a given model.
    """

    def __init__(self, alpha=0.2,     # Job separation rate
                        beta=0.98,    # Discount rate
                        gamma=0.7,    # Job offer rate
                        c=6.0,        # Unemployment compensation
                        sigma=2.0,    # Utility parameter
                        w_vec=None,   # Possible wage values
                        p_vec=None): # Probabilities over w_vec

        self.alpha, self.beta, self.gamma, self.c = alpha, beta, gamma, c
        self.sigma = sigma

        # 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(alpha, beta, gamma, c, sigma, 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, sigma) + beta * ((1 - alpha) * V[w_idx] + alpha *␣
↪U)

    U_new = u(c, sigma) + beta * (1 - gamma) * U + \
                    beta * gamma * np.sum(np.maximum(U, V) * p_vec)

    return U_new


def solve_mccall_model(mcm, tol=1e-5, max_iter=2000):
    """
    Iterates to convergence on the Bellman equations

    Parameters
    ----------
    mcm : an instance of McCallModel
    tol : float
        error tolerance
    max_iter : int
        the maximum number of iterations
    """

    V = np.ones(len(mcm.w_vec))  # Initial guess of V
    V_new = np.empty_like(V)     # To store updates to V
    U = 1                        # Initial guess of U
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        U_new = _update_bellman(mcm.alpha, mcm.beta, mcm.gamma,
                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
```

The second piece of code repeated from *the McCall model lecture* is used to complete the reservation wage

```
def compute_reservation_wage(mcm, return_values=False):
    """
    Computes the reservation wage of an instance of the McCall model
    by finding the smallest w such that V(w) > U.

    If V(w) > U for all w, then the reservation wage w_bar is set to
    the lowest wage in mcm.w_vec.

    If v(w) < U for all w, then w_bar is set to np.inf.
```

```
    Parameters
    ----------
    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

```
from scipy.stats import norm
from scipy.optimize import brentq

# Some global variables that will stay constant
alpha  = 0.013
alpha_q = (1-(1-alpha)**3)    # quarterly (alpha is monthly)
b      = 0.0124
d      = 0.00822
beta   = 0.98
gamma  = 1.0
sigma  = 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)
    lmda = gamma * np.sum(p_vec[w_vec-tau > w_bar])
    return w_bar, lmda, 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, lmda, V, U = compute_optimal_quantities(c, tau)

    # Compute steady state employment and unemployment rates
    lm = LakeModel(alpha=alpha_q, lmda=lmda, b=b, d=d)
    x = lm.rate_steady_state()
    e, u = 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
```

```
if __name__ == '__main__':

    # 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))

    ax = axes[0, 0]
    ax.plot(c_vec, unempl_vec, 'b-', lw=2, alpha=0.7)
    ax.set_title('unemployment')

    ax = axes[0, 1]
    ax.plot(c_vec, empl_vec, 'b-', lw=2, alpha=0.7)
    ax.set_title('employment')

    ax = axes[1, 0]
    ax.plot(c_vec, tax_vec, 'b-', lw=2, alpha=0.7)
    ax.set_title('tax')

    ax = axes[1, 1]
    ax.plot(c_vec, welfare_vec, 'b-', lw=2, alpha=0.7)
    ax.set_title('welfare')

    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

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 α and λ follow *[DFH06]*)

Suppose that in response to new legislation the hiring rate reduces to $\lambda = 0.2$

Plot the transition dynamics of the unemployment and employment stocks for 50 periods

Plot the transition dynamics for the rates

How long does the economy take to converge to its new steady state?

What is the new steady state level of employment?

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?

Solutions

Lake Model Solutions

Excercise 1

We begin by initializing the variables and import the necessary modules

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

import matplotlib
matplotlib.style.use('ggplot')
```

Construct the class containing the default parameters and record the steady state.

```
lm = LakeModel()
x0 = lm.rate_steady_state()
print("Initial Steady State: %s" % x0)
```

```
Initial Steady State: [ 0.91733194  0.08266806]
```

Initialize the simulation values

```
N0 = 100
T = 50
```

New legislation changes λ to 0.2

```
lm.lmda = 0.2

xbar = lm.rate_steady_state() # new steady state
X_path = np.vstack(lm.simulate_stock_path(x0 * N0, T))
x_path = np.vstack(lm.simulate_rate_path(x0, T))
print("New Steady State: %s" % xbar)
```

```
New Steady State: [ 0.88690427  0.11309573]
```

Now plot stocks

```
fig, axes = plt.subplots(3, 1, figsize=[10,9])
ax = axes[0]
ax.plot(X_path[:,0], 'b-')
ax.set_title('employment')

ax = axes[1]
ax.plot(X_path[:,1], 'b-')
ax.set_title('unemployment')

ax = axes[2]
ax.plot(X_path.sum(1), 'b-')
ax.set_title('labor force')

plt.tight_layout()
plt.show()
```

And how the rates evolve:

```
fig, axes = plt.subplots(2, 1, figsize=(10, 8))

ax = axes[0]
ax.plot(x_path[:,0], 'b-')
ax.hlines(xbar[0],0,T,'r','--')
ax.set_title('employment rate')

ax = axes[1]
ax.plot(x_path[:,1], 'b-')
ax.hlines(xbar[1], 0, T, 'r','--')
ax.set_title('unemployment rate')

plt.tight_layout()
plt.show()
```

We see that it takes 20 periods for the economy to converge to it's new steady state levels

Exercise 2

This next exercise has the economy expriencing a boom in entrances to the labor market and then later returning to the original levels. For 20 periods the economy has a new entry rate into the labor market

Let's start off at the baseline parameterization and record the steady state:

```
lm = LakeModel()
x0 = lm.rate_steady_state()
```

Here are the other parameters:

```
b_hat = 0.003
T_hat = 20
```

Let's increase b to the new value and simulate for 20 periods

```
lm.b = b_hat
X_path1 = np.vstack(lm.simulate_stock_path(x0 * N0, T_hat)) # simulate stocks
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

```
lm.b = 0.0124
X_path2 = np.vstack(lm.simulate_stock_path(X_path1[-1,:2],T-T_hat+1)) #␣
↪simulate stocks
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

```
x_path = np.vstack([x_path1,x_path2[1:]]) # note [1:] to avoid doubling␣
↪period 20
X_path = np.vstack([X_path1,X_path2[1:]])

fig, axes = plt.subplots(3, 1, figsize=[10,9])

ax = axes[0]
ax.plot(X_path[:,0], 'b-')
ax.set_title('employment')

ax = axes[1]
ax.plot(X_path[:,1], 'b-')
ax.set_title('unemployment')

ax = axes[2]
ax.plot(X_path.sum(1), 'b-')
ax.set_title('labor Force')

plt.tight_layout()
plt.show()
```

And the rates:

```
fig, axes = plt.subplots(2, 1, figsize=[10,6])

ax = axes[0]
ax.plot(x_path[:,0], 'b-')
ax.hlines(x0[0],0,T,'r','--')
ax.set_title('employment Rate')

ax = axes[1]
ax.plot(x_path[:,1], 'b-')
ax.hlines(x0[1],0,T,'r','--')
ax.set_title('unemployment Rate')

plt.tight_layout()
plt.show()
```

Uncertainty Traps

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 entrepreuneurs – 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

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

$$\theta_{t+1} = \rho\theta_t + \sigma_\theta w_{t+1}$$

where

- $\sigma_\theta > 0$ and $0 < \rho < 1$
- $\{w_t\}$ is IID and standard normal

The random variable θ_t is not observable at any time

Output

There is a total $\bar{M}$ of risk averse entrepreneurs

Output of the m-th entrepreneur, conditional on being active in the market at time t, is equal to

$$x_m = \theta + \epsilon_m \quad \text{where} \quad \epsilon_m \sim N\left(0, \gamma_x^{-1}\right) \tag{6.2}$$

Here the time subscript has been dropped to simplify notation

The inverse of the shock variance, γ_x, is called the shock's **precision**

The higher is the precision, the more informative x_m is about the fundamental

Output shocks are independent across time and firms

Information and Beliefs

All entrepreneurs start with identical beliefs about θ_0

Signals are publicly observable and hence all agents have identical beliefs always

Dropping time subscripts, beliefs for current θ are represented by the normal distribution $N(\mu, \gamma^{-1})$

Here γ is the precision of beliefs; its inverse is the degree of uncertainty

These parameters are updated by Kalman filtering

Let

- $\mathbb{M} \subset \{1, \ldots, \bar{M}\}$ 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

$$\mu' = \rho \frac{\gamma\mu + M\gamma_x X}{\gamma + M\gamma_x} \tag{6.3}$$

$$\gamma' = \left(\frac{\rho^2}{\gamma + M\gamma_x} + \sigma_\theta^2\right)^{-1} \tag{6.4}$$

These are standard Kalman filtering results applied to the current setting

Exercise 1 provides more details on how (6.3) and (6.4) are derived, and then asks you to fill in remaining steps

The next figure plots the law of motion for the precision in (6.4) 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 we'll see, the number of active firms fluctuates stochastically

Participation

Omitting time subscripts once more, entrepreneurs enter the market in the current period if

$$\mathbb{E}[u(x_m - F_m)] > c \tag{6.5}$$

Here

- the mathematical expectation of x_m is based on (6.2) and beliefs $N(\mu, \gamma^{-1})$ for θ

- 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 (6.5)

The utility function has the constant absolute risk aversion form

$$u(x) = \frac{1}{a}\left(1 - \exp(-ax)\right) \tag{6.6}$$

where a is a positive parameter

Combining (6.5) and (6.6), entrepreneur m participates in the market (or is said to be active) when

$$\frac{1}{a}\left\{1 - \mathbb{E}[\exp\left(-a(\theta + \epsilon_m - F_m)\right)]\right\} > c$$

Using standard formulas for expectations of lognormal random variables, this is equivalent to the condition

$$\psi(\mu, \gamma, F_m) := \frac{1}{a}\left(1 - \exp\left(-a\mu + aF_m + \frac{a^2\left(\frac{1}{\gamma} + \frac{1}{\gamma_x}\right)}{2}\right)\right) - c > 0 \tag{6.7}$$

Implementation

We want to simulate this economy

As a first step, let's put together a class that bundles

- the parameters, the current value of θ and the current values of the two belief parameters μ and γ
- methods to update θ, μ and γ, as well as to determine the number of active firms and their outputs

The updating methods follow the laws of motion for θ, μ and γ given above

The method to evaluate the number of active firms generates $F_1, \ldots, F_{\bar{M}}$ and tests condition (6.7) for each firm

The *__init__* method encodes as default values the parameters we'll use in the simulations below

Here's the code, which can be obtained from GitHub

```
from __future__ import division
import numpy as np

class UncertaintyTrapEcon(object):

    def __init__(self,
                 a=1.5,           # Risk aversion
                 gx=0.5,          # Production shock precision
                 rho=0.99,        # Correlation coefficient for theta
                 sig_theta=0.5,   # Std dev of theta shock
                 num_firms=100,   # Number of firms
                 sig_F=1.5,       # Std dev of fixed costs
```

```
                c=-420,          # External opportunity cost
                mu_init=0,       # Initial value for mu
                gamma_init=4,    # Initial value for gamma
                theta_init=0):   # Initial value for theta

        # == Record values == #
        self.a, self.gx, self.rho, self.sig_theta = a, gx, rho, sig_theta
        self.num_firms, self.sig_F, self.c, = num_firms, sig_F, c
        self.sd_x = np.sqrt(1/ gx)

        # == Initialize states == #
        self.gamma, self.mu, self.theta =  gamma_init, mu_init, theta_init

    def psi(self, F):
        temp1 = -self.a * (self.mu - F)
        temp2 = self.a**2 * (1/self.gamma + 1/self.gx) / 2
        return (1 / self.a) * (1 - np.exp(temp1 + temp2)) - self.c

    def update_beliefs(self, X, M):
        """
        Update beliefs (mu, gamma) based on aggregates X and M.
        """
        # Simplify names
        gx, rho, sig_theta = self.gx, self.rho, self.sig_theta
        # Update mu
        temp1 = rho * (self.gamma * self.mu + M * gx * X)
        temp2 = self.gamma + M * gx
        self.mu =  temp1 / temp2
        # Update gamma
        self.gamma = 1 / (rho**2 / (self.gamma + M * gx) + sig_theta**2)

    def update_theta(self, w):
        """
        Update the fundamental state theta given shock w.
        """
        self.theta = self.rho * self.theta + self.sig_theta * w

    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.sig_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.sd_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

Results

Let's look first at the dynamics of μ, which the agents use to track θ

We see that μ tracks θ well when there are sufficient firms in the market

However, there are times when μ tracks θ poorly due to insufficient information

These are episodes where the uncertainty traps take hold

During these episodes

- precision is low and uncertainty is high
- few firms are in the market

To get a clearer idea of the dynamics, let's look at all the main time series at once, for a given set of shocks

Notice how the traps only take hold after a sequence of bad draws for the fundamental

Thus, the model gives us a *propagation mechanism* that maps bad random draws into long downturns in economic activity

Exercises

Exercise 1

Fill in the details behind (6.3) and (6.4) based on the following standard result (see, e.g., p. 24 of *[YS05]*)

Fact Let $\mathbf{x} = (x_1, \ldots, x_M)$ be a vector of IID draws from common distribution $N(\theta, 1/\gamma_x)$ and let $\bar{x}$ be the sample mean. If γ_x is known and the prior for θ is $N(\mu, 1/\gamma)$, then the posterior distribution of θ given $\mathbf{x}$ is

$$\pi(\theta \,|\, \mathbf{x}) = N(\mu_0, 1/\gamma_0)$$

where

$$\mu_0 = \frac{\mu\gamma + M\bar{x}\gamma_x}{\gamma + M\gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M\gamma_x$$

Exercise 2

Modulo randomness, replicate the simulation figures shown above

- Use the parameter values listed as defaults in the *__init__* method of the *UncertaintyTrapEcon* class

Solutions

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import itertools
```

Exercise 1

This exercise asked you to validate the laws of motion for γ and μ given in the lecture, based on the stated result about Bayesian updating in a scalar Gaussian setting. The stated result tells us that after observing average output X of the M firms, our posterior beliefs will be

$$N(\mu_0, 1/\gamma_0)$$

where

$$\mu_0 = \frac{\mu\gamma + MX\gamma_x}{\gamma + M\gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M\gamma_x$$

If we take a random variable θ with this distribution and then evaluate the distribution of $\rho\theta + \sigma_\theta w$ where w is independent and standard normal, we get the expressions for μ' and γ' given in the lecture.

Exercise 2

First let's replicate the plot that illustrates the law of motion for precision, which is

$$\gamma_{t+1} = \left(\frac{\rho^2}{\gamma_t + M\gamma_x} + \sigma_\theta^2 \right)^{-1}$$

Here M is the number of active firms. The next figure plots γ_{t+1} against γ_t on a 45 degree diagram for different values of M

```
palette = itertools.cycle(sns.color_palette())
econ = UncertaintyTrapEcon()
rho, sig_theta, gx = econ.rho, econ.sig_theta, econ.gx # simplify names
g = np.linspace(1e-10, 3, 200)  # gamma grid
fig, ax = plt.subplots(figsize=(9, 9))
ax.plot(g, g, 'k-') # 45 degree line

for M in range(7):
    g_next = 1 / (rho**2 / (g + M * gx) + sig_theta**2)
    label_string = r"$M = {}$".format(M)
    ax.plot(g, g_next, lw=2, label=label_string, color=next(palette))
ax.legend(loc='lower right', fontsize=14)
ax.set_xlabel(r'$\gamma$', fontsize=16)
ax.set_ylabel(r"$\gamma'$", fontsize=16)
ax.grid()
plt.show()
```

The points where the curves hit the 45 degree lines are the long run steady states corresponding to each M, if that value of M was to remain fixed. As the number of firms falls, so does the long run steady state of precision.

Next let's generate time series for beliefs and the aggregates – that is, the number of active firms and average output.

```
sim_length=2000

mu_vec = np.empty(sim_length)
theta_vec = np.empty(sim_length)
gamma_vec = np.empty(sim_length)
X_vec = np.empty(sim_length)
M_vec = np.empty(sim_length)
```

```
mu_vec[0] = econ.mu
gamma_vec[0] = econ.gamma
theta_vec[0] = 0

w_shocks = np.random.randn(sim_length)

for t in range(sim_length-1):
    X, M = econ.gen_aggregates()
    X_vec[t] = X
    M_vec[t] = M

    econ.update_beliefs(X, M)
    econ.update_theta(w_shocks[t])

    mu_vec[t+1] = econ.mu
    gamma_vec[t+1] = econ.gamma
    theta_vec[t+1] = econ.theta

# Record final values of aggregates
X, M = econ.gen_aggregates()
X_vec[-1] = X
M_vec[-1] = M
```

First let's see how well μ tracks θ in these simulations

```
fig, ax = plt.subplots(figsize=(9, 6))
ax.plot(range(sim_length), theta_vec, alpha=0.6, lw=2, label=r"$\theta$")
ax.plot(range(sim_length), mu_vec, alpha=0.6, lw=2, label=r"$\mu$")
ax.legend(fontsize=16)
plt.show()
```

Now let's plot the whole thing together

```
fig, axes = plt.subplots(4, 1, figsize=(12, 20))
# Add some spacing
fig.subplots_adjust(hspace=0.3)

series = (theta_vec, mu_vec, gamma_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("time series for {}".format(name), fontsize=16)
    ax.grid()

plt.show()
```

time series for θ
5
0
−5
−10
0
250
500
750
1000
1250
1500
1750
2000

time series for μ
8
6
4
2
0
−2
−4
−6
−8
0
250
500
750
1000
1250
1500
1750
2000

time series for γ
4
3
2
1
0
0
250
500
750
1000
1250
1500
1750
2000

time series for M
100
80
60
20

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.)

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?"

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 agent's problem differs from a planner's, and how a planning problem can be used to compute rational expectations quantities

We will also learn about how a rational expectations equilibrium can be characterized as a fixed point of a mapping from a *perceived law of motion* to an *actual law of motion*

Equality between a perceived and an actual law of motion for endogenous market-wide objects captures in a nutshell what the rational expectations equilibrium concept is all about

Finally, we will learn about the important "Big K, little k" trick, a modeling device widely used in macroeconomics

Except that for us

- Instead of "Big K" it will be "Big Y"
- Instead of "little k" it will be "little y"

The Big Y, little y trick

This widely used method applies in contexts in which a "representative firm" or agent is a "price taker" operating within a competitive equilibrium

We want to impose that

- The representative firm or individual takes *aggregate* Y as given when it chooses individual y, but ...
- At the end of the day, $Y = y$, so that the representative firm is indeed representative

The Big Y, little y trick accomplishes these two goals by

- Taking Y as beyond control when posing the choice problem of who chooses y; but ...
- Imposing $Y = y$ *after* having solved the individual's optimization problem

Please watch for how this strategy is applied as the lecture unfolds

We begin by applying the Big Y, little y trick in a very simple static context

A simple static example of the Big Y, little y trick

Consider a static model in which a collection of n firms produce a homogeneous good that is sold in a competitive market

Each of these n firms sells output y

The price p of the good lies on an inverse demand curve

$$p = a_0 - a_1 Y \tag{6.8}$$

where

- $a_i > 0$ for $i = 0, 1$
- $Y = ny$ is the market-wide level of output

Each firm has total cost function

$$c(y) = c_1 y + 0.5 c_2 y^2, \qquad c_i > 0 \text{ for } i = 1, 2$$

The profits of a representative firm are $py - c(y)$

Using (6.8), we can express the problem of the representative firm as

$$\max_y \Big[(a_0 - a_1 Y) y - c_1 y - 0.5 c_2 y^2 \Big] \tag{6.9}$$

In posing problem (6.9), 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 (6.9)

This assures that the firm is a price taker

The first order condition for problem (6.9) is

$$a_0 - a_1 Y - c_1 - c_2 y = 0 \tag{6.10}$$

At this point, *but not before*, we substitute $Y = ny$ into (6.10) to obtain the following linear equation

$$a_0 - c_1 - (a_1 + n^{-1} c_2) Y = 0 \tag{6.11}$$

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 (6.8)

Further Reading

References for this lecture include

- *[LP71]*
- *[Sar87]*, chapter XIV
- *[LS12]*, chapter 7

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

$$p_t = a_0 - a_1 Y_t \tag{6.12}$$

where

- $a_i > 0$ for $i = 0, 1$
- $Y_t = n y_t$ is the market-wide level of output

The Firm's Problem

Each firm is a price taker

While it faces no uncertainty, it does face adjustment costs

In particular, it chooses a production plan to maximize

$$\sum_{t=0}^{\infty} \beta^t r_t \tag{6.13}$$

where

$$r_t := p_t y_t - \frac{\gamma (y_{t+1} - y_t)^2}{2}, \qquad y_0 \text{ given} \tag{6.14}$$

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 at time t

To state the firm's optimization problem completely requires that we specify dynamics for all state variables

This includes ones that the firm cares about but does not control like p_t

We turn to this problem now

Prices and Aggregate Output

In view of (6.12), the firm's incentive to forecast the market price translates into an incentive to forecast aggregate output Y_t

Aggregate output depends on the choices of other firms

We assume that n is such a large number that the output of any single firm has a negligible effect on aggregate output

That justifies firms in regarding their forecasts of aggregate output as being unaffected by their own output decisions

The Firm's Beliefs

We suppose the firm believes that market-wide output Y_t follows the law of motion

$$Y_{t+1} = H(Y_t) \tag{6.15}$$

where Y_0 is a known initial condition

The *belief function* H is an equilibrium object, and hence remains to be determined

Optimal Behavior Given Beliefs

For now let's fix a particular belief H in (6.15) and investigate the firm's response to it

Let v be the optimal value function for the firm's problem given H

The value function satisfies the Bellman equation

$$v(y, Y) = \max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \tag{6.16}$$

Let's denote the firm's optimal policy function by h, so that

$$y_{t+1} = h(y_t, Y_t) \tag{6.17}$$

where

$$h(y, Y) := \arg\max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \tag{6.18}$$

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

$$-\gamma(y' - y) + \beta v_y(y', H(Y)) = 0 \tag{6.19}$$

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 (6.16), giving

$$v_y(y, Y) = a_0 - a_1 Y + \gamma(y' - y)$$

Substituting this equation into (6.19) gives the *Euler equation*

$$-\gamma(y_{t+1} - y_t) + \beta[a_0 - a_1 Y_{t+1} + \gamma(y_{t+2} - y_{t+1})] = 0 \tag{6.20}$$

The firm optimally sets an output path that satisfies (6.20), taking (6.15) as given, and subject to

- the initial conditions for (y_0, Y_0)
- the terminal condition $\lim_{t\to\infty} \beta^t y_t v_y(y_t, Y_t) = 0$

This last condition is called the *transversality condition*, and acts as a first-order necessary condition "at infinity"

The firm's decision rule solves the difference equation (6.20) subject to the given initial condition y_0 and the transversality condition

Note that solving the Bellman equation (6.16) for v and then h in (6.18) yields a decision rule that automatically imposes both the Euler equation (6.20) 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

$$Y_{t+1} = nh(Y_t/n, Y_t) \tag{6.21}$$

Thus, when firms believe that the law of motion for market-wide output is (6.15), their optimizing behavior makes the actual law of motion be (6.21)

Definition of Rational Expectations Equilibrium

A *rational expectations equilibrium* or *recursive competitive equilibrium* of the model with adjustment costs is a decision rule h and an aggregate law of motion H such that

1. Given belief H, the map h is the firm's optimal policy function
2. The law of motion H satisfies $H(Y) = nh(Y/n, Y)$ for all Y

Thus, a rational expectations equilibrium equates the perceived and actual laws of motion (6.15) and (6.21)

Fixed point characterization

As we've seen, the firm's optimum problem induces a mapping Φ from a perceived law of motion H for market-wide output to an actual law of motion $\Phi(H)$

The mapping Φ is the composition of two operations, taking a perceived law of motion into a decision rule via (6.16)–(6.18), and a decision rule into an actual law via (6.21)

The H component of a rational expectations equilibrium is a fixed point of Φ

Computation of an Equilibrium

Now let's consider the problem of computing the rational expectations equilibrium

Misbehavior of Φ

Readers accustomed to dynamic programming arguments might try to address this problem by choosing some guess H_0 for the aggregate law of motion and then iterating with Φ

Unfortunately, the mapping Φ is not a contraction

In particular, there is no guarantee that direct iterations on Φ converge[1]

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 we'll see, this planning problem can be solved by LQ control (*linear regulator*)

The optimal quantities from the planning problem are rational expectations equilibrium quantities

The rational expectations equilibrium price can be obtained as a shadow price in the planning problem

For convenience, in this section we set $n = 1$

We first compute a sum of consumer and producer surplus at time t

$$s(Y_t, Y_{t+1}) := \int_0^{Y_t} (a_0 - a_1 x)\, dx - \frac{\gamma(Y_{t+1} - Y_t)^2}{2} \tag{6.22}$$

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

[1] A literature that studies whether models populated with agents who learn can converge to rational expectations equilibria features iterations on a modification of the mapping Φ that can be approximated as $\gamma\Phi + (1-\gamma)I$. Here I is the identity operator and $\gamma \in (0,1)$ is a *relaxation parameter*. See *[MS89]* and *[EH01]* for statements and applications of this approach to establish conditions under which collections of adaptive agents who use least squares learning converge to a rational expectations equilibrium.

Solution of the Planning Problem

Evaluating the integral in (6.22) yields the quadratic form $a_0 Y_t - a_1 Y_t^2 / 2$

As a result, the Bellman equation for the planning problem is

$$V(Y) = \max_{Y'} \left\{ a_0 Y - \frac{a_1}{2} Y^2 - \frac{\gamma (Y' - Y)^2}{2} + \beta V(Y') \right\} \tag{6.23}$$

The associated first order condition is

$$-\gamma (Y' - Y) + \beta V'(Y') = 0 \tag{6.24}$$

Applying the same Benveniste-Scheinkman formula gives

$$V'(Y) = a_0 - a_1 Y + \gamma (Y' - Y)$$

Substituting this into equation (6.24) and rearranging leads to the Euler equation

$$\beta a_0 + \gamma Y_t - [\beta a_1 + \gamma (1 + \beta)] Y_{t+1} + \gamma \beta Y_{t+2} = 0 \tag{6.25}$$

The Key Insight

Return to equation (6.20) 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 (6.25) and (6.20) are identical

Thus, the Euler equation for the planning problem matches the second-order difference equation that we derived by

1. finding the Euler equation of the representative firm and
2. substituting into it the expression $Y_t = n y_t$ that "makes the representative firm be representative"

If it is appropriate to apply the same terminal conditions for these two difference equations, which it is, then we have verified that a solution of the planning problem is also a rational expectations equilibrium quantity sequence

It follows that for this example we can compute equilibrium quantities by forming the optimal linear regulator problem corresponding to the Bellman equation (6.23)

The optimal policy function for the planning problem is the aggregate law of motion H that the representative firm faces within a rational expectations equilibrium.

Structure of the Law of Motion

As you are asked to show in the exercises, the fact that the planner's problem is an LQ problem implies an optimal policy — and hence aggregate law of motion — taking the form

$$Y_{t+1} = \kappa_0 + \kappa_1 Y_t \tag{6.26}$$

for some parameter pair κ_0, κ_1

Now that we know the aggregate law of motion is linear, we can see from the firm's Bellman equation (6.16) that the firm's problem can also be framed as an LQ problem

As you're asked to show in the exercises, the LQ formulation of the firm's problem implies a law of motion that looks as follows

$$y_{t+1} = h_0 + h_1 y_t + h_2 Y_t \tag{6.27}$$

Hence a rational expectations equilibrium will be defined by the parameters $(\kappa_0, \kappa_1, h_0, h_1, h_2)$ in (6.26)–(6.27)

Exercises

Exercise 1

Consider the firm problem *described above*

Let the firm's belief function H be as given in (6.26)

Formulate the firm's problem as a discounted optimal linear regulator problem, being careful to describe all of the objects needed

Use the class `LQ` from the QuantEcon.py package to solve the firm's problem for the following parameter values:

$$a_0 = 100, a_1 = 0.05, \beta = 0.95, \gamma = 10, \kappa_0 = 95.5, \kappa_1 = 0.95$$

Express the solution of the firm's problem in the form (6.27) and give the values for each h_j

If there were n identical competitive firms all behaving according to (6.27), what would (6.27) imply for the *actual* law of motion (6.15) for market supply

Exercise 2

Consider the following κ_0, κ_1 pairs as candidates for the aggregate law of motion component of a rational expectations equilibrium (see (6.26))

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 planner's problem *described above*

1. Formulate the planner's problem as an LQ problem
2. Solve it using the same parameter values in exercise 1
   - $a_0 = 100, a_1 = 0.05, \beta = 0.95, \gamma = 10$
3. Represent the solution in the form $Y_{t+1} = \kappa_0 + \kappa_1 Y_t$
4. Compare your answer with the results from exercise 2

Exercise 4

A monopolist faces the industry demand curve (6.12) 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.

Solutions

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

We'll 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

$$x_t = \begin{bmatrix} y_t \\ Y_t \\ 1 \end{bmatrix}, \qquad u_t = y_{t+1} - y_t$$

For , B, Q, R we set

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \kappa_1 & \kappa_0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad R = \begin{bmatrix} 0 & a_1/2 & -a_0/2 \\ a_1/2 & 0 & 0 \\ -a_0/2 & 0 & 0 \end{bmatrix}, \quad Q = \gamma/2$$

By multiplying out you can confirm that

- $x_t' R x_t + u_t' Q u_t = -r_t$
- $x_{t+1} = A x_t + B u_t$

We'll use the module `lqcontrol.py` to solve the firm's problem at the stated parameter values

This will return an LQ policy F with the interpretation $u_t = -F x_t$, or

$$y_{t+1} - y_t = -F_0 y_t - F_1 Y_t - F_2$$

Matching parameters with $y_{t+1} = h_0 + h_1 y_t + h_2 Y_t$ leads to

$$h_0 = -F_2, \quad h_1 = 1 - F_0, \quad h_2 = -F_1$$

Here's our solution

```
# == Model parameters == #

a0      = 100
a1      = 0.05
beta    = 0.95
gamma   = 10.0

# == Beliefs == #

kappa0  = 95.5
kappa1  = 0.95

# == Formulate the LQ problem == #

A = np.array([[1, 0, 0], [0, kappa1, kappa0], [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 * gamma

# == Solve for the optimal policy == #

lq = LQ(Q, R, A, B, beta=beta)
P, F, d = lq.stationary_values()
F = F.flatten()
out1 = "F = [{0:.3f}, {1:.3f}, {2:.3f}]".format(F[0], F[1], F[2])
h0, h1, h2 = -F[2], 1 - F[0], -F[1]
out2 = "(h0, h1, h2) = ({0:.3f}, {1:.3f}, {2:.3f})".format(h0, h1, h2)

print(out1)
print(out2)
```

```
F = [-0.000, 0.046, -96.949]
(h0, h1, h2) = (96.949, 1.000, -0.046)
```

The implication is that

$$y_{t+1} = 96.949 + y_t - 0.046\,Y_t$$

For the case $n > 1$, recall that $Y_t = ny_t$, which, combined with the previous equation, yields

$$Y_{t+1} = n\,(96.949 + y_t - 0.046\,Y_t) = n96.949 + (1 - n0.046)Y_t$$

Exercise 2

To determine whether a κ_0, κ_1 pair forms the aggregate law of motion component of a rational expectations equilibrium, we can proceed as follows:

- Determine the corresponding firm law of motion $y_{t+1} = h_0 + h_1 y_t + h_2 Y_t$
- Test whether the associated aggregate law :$Y_{t+1} = nh(Y_t/n, Y_t)$ evaluates to $Y_{t+1} = \kappa_0 + \kappa_1 Y_t$

In the second step we can use $Y_t = ny_t = y_t$, so that $Y_{t+1} = nh(Y_t/n, Y_t)$ becomes

$$Y_{t+1} = h(Y_t, Y_t) = h_0 + (h_1 + h_2)Y_t$$

Hence to test the second step we can test $\kappa_0 = h_0$ and $\kappa_1 = h_1 + h_2$

The following code implements this test

```
candidates = (
          (94.0886298678, 0.923409232937),
          (93.2119845412, 0.984323478873),
          (95.0818452486, 0.952459076301)
             )

for kappa0, kappa1 in candidates:

    # == Form the associated law of motion == #
    A = np.array([[1, 0, 0], [0, kappa1, kappa0], [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((kappa0, kappa1), (h0, h1 + h2)):
        print('Equilibrium pair =', kappa0, kappa1)
        print('(h0, h1, h2) = ', h0, h1, h2)
        break
```

```
Equilibrium pair = 95.0818452486 0.952459076301
(h0, h1, h2) =  95.0818910013 1.0 -0.0475409444278
```

The output tells us that the answer is pair (iii), which implies $(h_0, h_1, h_2) = (95.0819, 1.0000, -.0475)$

(Notice we use `np.allclose` to test equality of floating point numbers, since exact equality is too strict)

Regarding the iterative algorithm, one could loop from a given (κ_0, κ_1) pair to the associated firm law and then to a new (κ_0, κ_1) pair

This amounts to implementing the operator Φ described in the lecture

(There is in general no guarantee that this iterative process will converge to a rational expectations equilibrium)

Exercise 3

We are asked to write the planner problem as an LQ problem

For the state and control vectors we choose

$$x_t = \begin{bmatrix} Y_t \\ 1 \end{bmatrix}, \quad u_t = Y_{t+1} - Y_t$$

For the LQ matrices we set

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad R = \begin{bmatrix} a_1/2 & -a_0/2 \\ -a_0/2 & 0 \end{bmatrix}, \quad Q = \gamma/2$$

By multiplying out you can confirm that

- $x_t' R x_t + u_t' Q u_t = -s(Y_t, Y_{t+1})$
- $x_{t+1} = A x_t + B u_t$

By obtaining the optimal policy and using $u_t = -F x_t$ or

$$Y_{t+1} - Y_t = -F_0 Y_t - F_1$$

we can obtain the implied aggregate law of motion via $\kappa_0 = -F_1$ and $\kappa_1 = 1 - F_0$

The Python code to solve this problem is below:

```
# == Formulate the planner's LQ problem == #

A = np.array([[1, 0], [0, 1]])
B = np.array([[1], [0]])
R = np.array([[a1 / 2, -a0 / 2], [-a0 / 2, 0]])
Q = 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()
kappa0, kappa1 = -F[1], 1 - F[0]
print(kappa0, kappa1)
```

```
95.0818745921 0.952459062704
```

The output yields the same (κ_0, κ_1) pair obtained as an equilibrium from the previous exercise

Exercise 4

The monopolist's LQ problem is almost identical to the planner's problem from the previous exercise, except that

$$R = \begin{bmatrix} a_1 & -a_0/2 \\ -a_0/2 & 0 \end{bmatrix}$$

The problem can be solved as follows

```
A = np.array([[1, 0], [0, 1]])
B = np.array([[1], [0]])
R = np.array([[a1, -a0 / 2], [-a0 / 2, 0]])
Q = gamma / 2

lq = LQ(Q, R, A, B, beta=beta)
P, F, d = lq.stationary_values()

F = F.flatten()
m0, m1 = -F[1], 1 - F[0]
print(m0, m1)
```

```
73.472944035 0.926527055965
```

We see that the law of motion for the monopolist is approximately $Y_{t+1} = 73.4729 + 0.9265 Y_t$

In the rational expectations case the law of motion was approximately $Y_{t+1} = 95.0818 + 0.9525 Y_t$

One way to compare these two laws of motion is by their fixed points, which give long run equilibrium output in each case

For laws of the form $Y_{t+1} = c_0 + c_1 Y_t$, the fixed point is $c_0/(1 - c_1)$

If you crunch the numbers, you will see that the monopolist adopts a lower long run quantity than obtained by the competitive market, implying a higher market price

This is analogous to the elementary static-case results

Markov Perfect Equilibrium

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 *[LS12]*

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 objectives

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 any given agent is affected by the decision rules of other agents

Individual payoff maximization requires that each agent solve a dynamic programming problem in response to this transition law

Markov perfect equilibrium is attained 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

$$p = a_0 - a_1(q_1 + q_2) \tag{6.28}$$

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 (6.28) and what follows,

- the time subscript is suppressed when possible to simplify notation
- $\hat{x}$ denotes a next period value of variable x

Each firm recognizes that its output affects total output and therefore the market price

The one-period payoff function of firm i is price times quantity minus adjustment costs:

$$\pi_i = pq_i - \gamma(\hat{q}_i - q_i)^2, \quad \gamma > 0, \tag{6.29}$$

Substituting the inverse demand curve (6.28) into (6.29) lets us express the one-period payoff as

$$\pi_i(q_i, q_{-i}, \hat{q}_i) = a_0 q_i - a_1 q_i^2 - a_1 q_i q_{-i} - \gamma(\hat{q}_i - q_i)^2, \tag{6.30}$$

where q_{-i} denotes the output of the firm other than i

The objective of the firm is to maximize $\sum_{t=0}^{\infty} \beta^t \pi_{it}$

Firm i chooses a decision rule that sets next period quantity $\hat{q}_i$ as a function f_i of the current state (q_i, q_{-i})

An essential aspect of a Markov perfect equilibrium is that each firm takes the decision rule of the other firm as known and given

Given f_{-i}, the Bellman equation of firm i is

$$v_i(q_i, q_{-i}) = \max_{\hat{q}_i} \{\pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i(\hat{q}_i, f_{-i}(q_{-i}, q_i))\} \tag{6.31}$$

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 (6.31)
- The maximizer on the right side of (6.31) 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 for all possible future states

This includes many states that will not be reached when we iterate forward on the pair of equilibrium strategies f_i

Computation

One strategy for computing a Markov perfect equilibrium is iterating to convergence on pairs of Bellman equations and decision rules

In particular, let v_i^j, f_i^j be the value function and policy function for firm i at the j-th iteration

Imagine constructing the iterates

$$v_i^{j+1}(q_i, q_{-i}) = \max_{\hat{q}_i} \left\{\pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i^j(\hat{q}_i, f_{-i}(q_{-i}, q_i))\right\} \tag{6.32}$$

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

Linear Markov perfect equilibria

As we saw in the duopoly example, the study of Markov perfect equilibria in games with two players leads us to an interrelated pair of Bellman equations

In linear quadratic dynamic games, these "stacked Bellman equations" become "stacked Riccati equations" with a tractable mathematical structure

We'll lay out that structure in a general setup and then apply it to some simple problems

A Coupled Linear Regulator Problem

We consider a general linear quadratic regulator game with two players

For convenience, we'll start with a finite horizon formulation, where t_0 is the initial date and t_1 is the common terminal date

Player i takes $\{u_{-it}\}$ as given and minimizes

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{ x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2x_t' W_i u_{it} + 2u_{-it}' M_i u_{it} \right\} \tag{6.33}$$

while the state evolves according to

$$x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t} \tag{6.34}$$

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 (6.33) and (6.34), then player 1's problem becomes minimization of

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{ x_t' \Pi_{1t} x_t + u_{1t}' Q_1 u_{1t} + 2 u_{1t}' \Gamma_{1t} x_t \right\} \tag{6.35}$$

subject to

$$x_{t+1} = \Lambda_{1t} x_t + B_1 u_{1t}, \tag{6.36}$$

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

$$F_{1t} = (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) \tag{6.37}$$

where P_{1t} is the solution of the matrix Riccati difference equation

$$P_{1t} = \Pi_{1t} - (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t})' (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) + \beta \Lambda_{1t}' P_{1t+1} \Lambda_{1t} \tag{6.38}$$

Similarly, the policy that solves player 2's problem is

$$F_{2t} = (Q_2 + \beta B_2' P_{2t+1} B_2)^{-1} (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) \tag{6.39}$$

where P_{2t} solves

$$P_{2t} = \Pi_{2t} - (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t})' (Q_2 + \beta B_2' P_{2t+1} B_2)^{-1} (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) + \beta \Lambda_{2t}' P_{2t+1} \Lambda_{2t} \tag{6.40}$$

Here in all cases $t = t_0, \ldots, t_1 - 1$ and the initial conditions are $P_{it_1} = 0$

The solution procedure is to use equations (6.37), (6.38), (6.39), and (6.40), and "work backwards" from time $t_1 - 1$

Since we're working backwards, P_{1t+1} and P_{2t+1} are taken as given at each stage

Moreover, since

- some terms on the right hand side of (6.37) contain F_{2t}
- some terms on the right hand side of (6.39) contain F_{1t}

we need to solve these $k_1 + k_2$ equations simultaneously

A key insight is that the equations (6.37) and (6.39) are linear in F_{1t} and F_{2t}

After these equations are solved, we can take the F_{it} and solve for P_{it} in (6.38) and (6.40)

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

Applications

Let's use these procedures to treat some applications, starting with the duopoly model

The duopoly case

To map the duopoly model into a coupled linear-quadratic dynamic programming problem, 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 (6.30)

The law of motion for the state x_t is $x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t}$ where

$$A := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B_1 := \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad B_2 := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The optimal decision rule of firm i will take the form $u_{it} = -F_i x_t$, inducing the following closed loop system for the evolution of x in the Markov perfect equilibrium:

$$x_{t+1} = (A - B_1 F_1 - B_1 F_2) x_t \tag{6.41}$$

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

```
"""
@authors: Chase Coleman, Thomas Sargent, John Stachurski

Markov Perfect Equilibrium for the simple duopoly example.

See the lecture at http://quant-econ.net/py/markov_perf.html for a
description of the model.
"""

from __future__ import division
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 = [[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

# == 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=beta)

# == Display policies == #
print("Computed policies for firm 1 and firm 2:\n")
print("F1 = {}".format(F1))
print("F2 = {}".format(F2))
print("\n")
```

Running the code produces the following output

```
run duopoly_mpe.py
```

```
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_2 as given is to use QuantEcon's *LQ* class

In particular, let's take *F2* as computed above, plug it into (6.35) and (6.36) to get firm 1's problem and solve it using *LQ*

We hope that the resulting policy will agree with *F1* as computed above

```
Lambda1 = A - np.dot(B2, F2)
lq1 = qe.LQ(Q1, R1, Lambda1, B1, beta=beta)
P1_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

```
np.allclose(F1, F1_ih)
```

```
True
```

Dynamics

Let's now investigate the dynamics of price and output in this simple duopoly model under the MPE policies

Given our optimal policies $F1$ and $F2$, the state evolves according to (6.41)

The following program

- imports $F1$ and $F2$ from the previous program along with all parameters
- computes the evolution of x_t using (6.41)
- extracts and plots industry output $q_t = q_{1t} + q_{2t}$ and price $p_t = a_0 - a_1 q_t$

```
import matplotlib.pyplot as plt

AF = A - B1.dot(F1) - B2.dot(F2)
n = 20
x = np.empty((3, n))
x[:, 0] = 1, 1, 1
for t in range(n-1):
    x[:, t+1] = np.dot(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

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's *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

- $C_{it} = c_{i1} + c_{i2} I_{it} + 0.5 c_{i3} I_{it}^2$
- $E_{it} = e_{i1} + e_{i2} q_{it} + 0.5 e_{i3} q_{it}^2$ where e_{ij}, c_{ij} are positive scalars

Inventories obey the laws of motion

$$I_{i,t+1} = (1-\delta) I_{it} + q_{it} - S_{it}$$

Demand is governed by the linear schedule

$$S_t = D p_{it} + b$$

where

- $S_t = \begin{bmatrix} S_{1t} & S_{2t} \end{bmatrix}'$
- D is a 2×2 negative definite matrix and
- b is a vector of constants

Firm i maximizes the undiscounted sum

$$\lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T} (p_{it} S_{it} - E_{it} - C_{it})$$

We can convert this to a linear quadratic problem by taking

$$u_{it} = \begin{bmatrix} p_{it} \\ q_{it} \end{bmatrix} \quad \text{and} \quad x_t = \begin{bmatrix} I_{1t} \\ I_{2t} \\ 1 \end{bmatrix}$$

Decision rules for price and quantity take the form $u_{it} = -F_i x_t$

The Markov perfect equilibrium of Judd's model can be computed by filling in the matrices appropriately

The exercise is to calculate these matrices and compute the following figures

The first figure shows the dynamics of inventories for each firm when the parameters are

```
delta   = 0.02
D       = np.array([[-1, 0.5], [0.5, -1]])
b       = np.array([25, 25])
c1 = c2 = np.array([1, -2, 1])
e1 = e2 = np.array([10, 10, 3])
```

Inventories trend to a common steady state

If we increase the depreciation rate to $\delta = 0.05$, then we expect steady state inventories to fall

This is indeed the case, as the next figure shows

Solutions

```
import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
from numpy import dot
```

Exercise 1

First let's compute the duopoly MPE under the stated parameters

```
# == 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 = [[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

# == 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=beta)
```

Now we evaluate the time path of industry output and prices given initial condition $q_{10} = q_{20} = 1$

```
AF = A - B1.dot(F1) - B2.dot(F2)
n = 20
x = np.empty((3, n))
x[:, 0] = 1, 1, 1
for t in range(n-1):
    x[:, t+1] = np.dot(AF, x[:, t])
q1 = x[1, :]
q2 = x[2, :]
q = q1 + q2        # Total output, MPE
p = a0 - a1 * q    # Price, MPE
```

Next let's have a look at the monopoly solution

For the state and control we take

$$x_t = q_t - \bar{q} \quad \text{and} \quad u_t = q_{t+1} - q_t$$

To convert to an LQ problem we set

$$R = a_1 \quad \text{and} \quad Q = \gamma$$

in the payoff function $x_t' R x_t + u_t' Q u_t$ and

$$A = B = 1$$

in the law of motion $x_{t+1} = A x_t + B u_t$

We solve for the optimal policy $u_t = -F x_t$ and track the resulting dynamics of $\{q_t\}$, starting at $q_0 = 2.0$

```
R = a1
Q = gamma
A = B = 1
lq_alt = qe.LQ(Q, R, A, B, beta=beta)
P, F, d = lq_alt.stationary_values()
q_bar = a0 / (2.0 * a1)
qm = np.empty(n)
qm[0] = 2
x0 = qm[0] - q_bar
```

```
x = x0
for i in range(1, n):
    x = A * x - B * F * x
    qm[i] = float(x) + q_bar
pm = a0 - a1 * qm
```

Let's have a look at the different time paths

```
fig, axes = plt.subplots(2, 1, figsize=(9, 9))

ax = axes[0]
ax.plot(qm, 'b-', lw=2, alpha=0.75, label='monopolist output')
ax.plot(q, 'g-', lw=2, alpha=0.75, label='MPE total output')
ax.set_ylabel("output")
ax.set_xlabel("time")
ax.set_ylim(2, 4)
ax.legend(loc='upper left', frameon=0)


ax = axes[1]
ax.plot(pm, 'b-', lw=2, alpha=0.75, label='monopolist price')
ax.plot(p, 'g-', lw=2, alpha=0.75, label='MPE price')
ax.set_ylabel("price")
ax.set_xlabel("time")
ax.legend(loc='upper right', frameon=0)
plt.show()
```

Exercise 2

We treat the case $\delta = 0.02$

```
delta   = 0.02
D       = np.array([[-1, 0.5], [0.5, -1]])
b       = np.array([25, 25])
c1 = c2 = np.array([1, -2, 1])
e1 = e2 = np.array([10, 10, 3])

delta_1 = 1 - delta
```

Recalling that the control and state are

$$u_{it} = \begin{bmatrix} p_{it} \\ q_{it} \end{bmatrix} \quad \text{and} \quad x_t = \begin{bmatrix} I_{1t} \\ I_{2t} \\ 1 \end{bmatrix}$$

we set up the matrices as follows:

```
# ==  Create matrices needed to compute the Nash feedback equilibrium == #

A = np.array([[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, 0],
                [0.5*c1[1], 0, c1[0]]])
R2 = -np.array([[0, 0, 0],
                [0, 0.5*c2[2], 0.5*c2[1]],
                [0, 0.5*c2[1], c2[0]]])

Q1 = np.array([[-0.5*e1[2], 0], [0, D[0, 0]]])
Q2 = np.array([[-0.5*e2[2], 0], [0, D[1, 1]]])

S1 = np.zeros((2, 2))
S2 = np.copy(S1)

W1 = np.array([[0, 0],
              [0, 0],
              [-0.5*e1[1], b[0]/2.]])
W2 = np.array([[0, 0],
              [0, 0],
              [-0.5*e2[1], b[1]/2.]])

M1 = np.array([[0, 0], [0, D[0, 1] / 2.]])
M2 = np.copy(M1)
```

We can now compute the equilibrium using `qe.nnash`

```
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2)

print("\nFirm 1's feedback rule:\n")
print(F1)

print("\nFirm 2's feedback rule:\n")
print(F2)
```

```
Firm 1's feedback rule:

[[  2.43666582e-01   2.72360627e-02  -6.82788293e+00]
 [  3.92370734e-01   1.39696451e-01  -3.77341073e+01]]

Firm 2's feedback rule:

[[  2.72360627e-02   2.43666582e-01  -6.82788293e+00]
 [  1.39696451e-01   3.92370734e-01  -3.77341073e+01]]
```

Now let's look at the dynamics of inventories, and reproduce the graph corresponding to $\delta = 0.02$

```
AF = A - B1.dot(F1) - B2.dot(F2)
n = 25
x = np.empty((3, n))
x[:, 0] = 2, 0, 1
for t in range(n-1):
    x[:, t+1] = np.dot(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(r'$\delta = {}$'.format(delta))
ax.legend()
plt.show()
```

Introduction to Asset Pricing

Contents

- *Introduction to Asset Pricing*
  - *Overview*
  - *Pricing Models*
  - *Prices in the Risk Neutral Case*
  - *Asset Prices under Risk Aversion*
  - *Implementation*
  - *Exercises*
  - *Solutions*

"A little knowledge of geometric series goes a long way" – Robert E. Lucas, Jr.

"Asset pricing is all about covariances" – Lars Peter Hansen

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

Pricing Models

In what follows let $\{d_t\}_{t \geq 0}$ be a stream of dividends

- A time-t **cum-dividend** asset is a claim to the stream $d_t, d_{t+1}, \ldots$
- A time-t **ex-dividend** asset is a claim to the stream $d_{t+1}, d_{t+2}, \ldots$

Let's look at some equations that we expect to hold for prices of assets under cum-dividend and ex-dividend contracts respectively

Risk Neutral Pricing

Our first scenario is risk-neutral pricing

Let $\beta = 1/(1+\rho)$ be an intertemporal discount factor, where ρ is the rate at which agents discount the future

The basic risk-neutral asset pricing equation for pricing one unit of a cum-dividend asset is

$$p_t = d_t + \beta \mathbb{E}_t[p_{t+1}] \tag{6.42}$$

This is a simple "cost equals expected benefit" relationship

Here $\mathbb{E}_t[y]$ denotes the best forecast of y, conditioned on information available at time t

For an ex-dividend asset, the basic risk-neutral asset pricing equation is

$$p_t = \beta \mathbb{E}_t[d_{t+1} + p_{t+1}] \tag{6.43}$$

Pricing with Random Discount Factor

What happens if for some reason traders discount payouts differently depending on the state of the world?

Michael Harrison and David Kreps *[HK79]* and Lars Peter Hansen and Scott Richard *[HR87]* showed that in quite general settings the price of an ex-dividend asset obeys

$$p_t = \mathbb{E}_t \left[m_{t+1}(d_{t+1} + p_{t+1}) \right] \tag{6.44}$$

for some **stochastic discount factor** m_{t+1}

The fixed discount factor β in (6.43) 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 $\mathrm{cov}_t(x_{t+1}, y_{t+1})$, we have

$$\mathbb{E}_t(x_{t+1} y_{t+1}) = \mathrm{cov}_t(x_{t+1}, y_{t+1}) + \mathbb{E}_t x_{t+1} \mathbb{E}_t y_{t+1} \tag{6.45}$$

If we apply this definition to the asset pricing equation (6.44) we obtain

$$p_t = \mathbb{E}_t m_{t+1} \mathbb{E}_t(d_{t+1} + p_{t+1}) + \mathrm{cov}_t(m_{t+1}, d_{t+1} + p_{t+1}) \tag{6.46}$$

It is useful to regard equation (6.46) as a generalization of equation (6.43)

- In equation (6.43), the stochastic discount factor $m_{t+1} = \beta$, a constant
- In equation (6.43), the covariance term $\mathrm{cov}_t(m_{t+1}, d_{t+1} + p_{t+1})$ is zero because $m_{t+1} = \beta$

Equation (6.46) 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 some expressions that this ratio should satisfy

For the case of an ex-dividend contract, we can divide both sides of (6.44) by d_t to get

$$v_t = \mathbb{E}_t \left[m_{t+1} \frac{d_{t+1}}{d_t} (1 + v_{t+1}) \right] \tag{6.47}$$

For the cum-dividend case, the corresponding expression is

$$v_t = 1 + \mathbb{E}_t \left[m_{t+1} \frac{d_{t+1}}{d_t} v_{t+1} \right] \tag{6.48}$$

Below we'll discuss the implications of these equations

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 (6.42) and iterating forward gives

$$\begin{aligned} p_t &= d + \beta p_{t+1} \\ &= d + \beta(d + \beta p_{t+2}) \\ &\vdots \\ &= d + \beta d + \beta^2 d + \cdots + \beta^{k-1} d + \beta^k p_{t+k} \end{aligned}$$

Unless prices explode in the future, this sequence converges to

$$\bar{p} := \frac{1}{1-\beta} d \tag{6.49}$$

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 $p_t = d + \beta p_{t+1}$

The ex-dividend equilibrium price is $(1-\beta)^{-1}\beta d$

Example 2: Dividends with deterministic growth paths

Consider a growing, non-random dividend process $d_{t+1} = g d_t$ where $0 < g\beta < 1$

While prices are not usually constant when dividends grow over time, the price dividend-ratio might be

If we guess this, substituting $v_t = v$ into (6.48) as well as our other assumptions, we get $v = 1 + \beta g v$

Since $\beta g < 1$, we have a unique positive solution for the cum-dividend case:

$$v = \frac{1}{1-\beta g}$$

The cum-dividend price is then

$$p_t = \frac{1}{1-\beta g} d_t \tag{6.50}$$

In view of (6.47), the ex-dividend formulas are

$$v = \frac{\beta g}{1-\beta g} \quad \text{and} \quad p_t = \frac{\beta g}{1-\beta g} d_t$$

If, in this example, we take $g = 1 + \kappa$ and let $\rho := 1/\beta - 1$, then the ex-dividend 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

$$d_{t+1} = g_{t+1} d_t \tag{6.51}$$

The stochastic growth factor $\{g_t\}$ is given by

$$g_t = g(X_t), \quad t = 1, 2, \ldots$$

where

1. $\{X_t\}$ is a finite Markov chain with state space S and transition probabilities

$$P(x, y) := \mathbb{P}\{X_{t+1} = y \,|\, X_t = x\} \qquad (x, y \in S)$$

1. g is a given function on S taking positive values

You can think of

- S as n possible "states of the world" and X_t as the current state
- g as a function that maps a given state X_t into a growth factor $g_t = g(X_t)$ for the endowment
- $\ln g_t = \ln(d_{t+1}/d_t)$ is the growth rate of dividends

(For a refresher on notation and theory for finite Markov chains see *this lecture*)

The next figure shows a simulation, where

- $\{X_t\}$ evolves as a discretized AR1 process produced using *Tauchen's method*
- $g_t = \exp(X_t)$, so that $\ln g_t = X_t$ is the growth rate

The code can be found here

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 $v_t = v(X_t)$

Staring with the cum-dividend case, we can substitute this guess into (6.48) to get

$$v(X_t) = 1 + \beta \mathbb{E}_t[g(X_{t+1}) v(X_{t+1})]$$

If we condition on $X_t = x$, this becomes

$$v(x) = 1 + \beta \sum_{y \in S} g(y) v(y) P(x, y)$$

or

$$v(x) = 1 + \beta \sum_{y \in S} K(x, y) v(y) \quad \text{where} \quad K(x, y) := g(y) P(x, y) \tag{6.52}$$

Suppose that there are n possible states $x_1, \ldots, x_n$

We can then think of (6.52) as n stacked equations, one for each state, and write it in matrix form as

$$v = \mathbb{1} + \beta K v \tag{6.53}$$

Here

- v is understood to be the column vector $(v(x_1), \ldots, v(x_n))'$
- K is the matrix $(K(x_i, x_j))_{1 \leq i,j \leq n}$
- $\mathbb{1}$ is a column vector of ones

When does (6.53) have a unique solution?

From the *Neumann series lemma* and Gelfand's formula, this will be the case if βK has spectral radius strictly less than one

In other words, we require that the eigenvalues of K be strictly less than β^{-1} in modulus

The solution is then

$$v = (I - \beta K)^{-1} \mathbb{1}$$

Similar reasoning in the ex-dividend case yields

$$v = (I - \beta K)^{-1} \beta K \mathbb{1} \tag{6.54}$$

Code

Let's calculate and plot the price-dividend ratio for the ex-dividend case 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

```
import numpy as np
import matplotlib.pyplot as plt
import quantecon as qe
from scipy.linalg import solve, eigvals

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()
ax.plot(mc.state_values, v, 'g-o', lw=2, alpha=0.7, label=r'$v$')
ax.set_ylabel("price-dividend ratio")
ax.set_xlabel("state")
ax.legend(loc='upper left')
plt.show()
```

Here's the figure it produces

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

Asset Prices under Risk Aversion

Now let's turn to the case where agents are risk averse

We'll price several distinct assets, including

- The price of an endowment stream
- A consol (a type of bond issued by the UK government in the 19th century)
- Call options on a consol

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

$$m_{t+1} = \beta \frac{u'(c_{t+1})}{u'(c_t)} \tag{6.55}$$

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 (6.51)

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, we'll assume the **constant relative risk aversion** (CRRA) specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \text{ with } \gamma > 0 \tag{6.56}$$

When $\gamma = 1$ we let $u(c) = \ln c$

Inserting the CRRA specification into (6.55) and using $c_t = d_t$ gives

$$m_{t+1} = \beta \left(\frac{c_{t+1}}{c_t} \right)^{-\gamma} = \beta g_{t+1}^{-\gamma} \tag{6.57}$$

Substituting this into (6.47) gives the ex-dividend price-dividend ratio formula

$$v(X_t) = \beta \mathbb{E}_t \left[g(X_{t+1})^{1-\gamma} (1 + v(X_{t+1})) \right]$$

Conditioning on $X_t = x$, we can write this as

$$v(x) = \beta \sum_{y \in S} g(y)^{1-\gamma}(1 + v(y))P(x, y)$$

If we let

$$J(x, y) := g(y)^{1-\gamma}P(x, y)$$

then we can rewrite in vector form as

$$v = \beta J(\mathbb{1} + v)$$

Assuming that the spectral radius of J is strictly less than β^{-1}, this equation has the unique solution

$$v = (I - \beta J)^{-1}\beta J\mathbb{1} \tag{6.58}$$

Here's a plot of v as a function of the state for several values of γ, with a positively correlated Markov process and $g(x) = \exp(x)$

The code with all details can be found here

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 (6.57), higher growth decreases the discount factor, lowering the weight placed on future returns

Special cases

In the special case $\gamma = 1$, we have $J = P$

Recalling that $P^i \mathbb{1} = \mathbb{1}$ for all i and applying *Neumann's geometric series lemma*, we are led to

$$v = \beta(I - \beta P)^{-1} \mathbb{1} = \beta \sum_{i=0}^{\infty} \beta^i P^i \mathbb{1} = \beta \frac{1}{1-\beta} \mathbb{1}$$

Thus, with log preferences, the price-dividend ratio for a Lucas tree is constant

Alternatively, if $\gamma = 0$, then $J = K$ and we recover the risk neutral solution (6.54)

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

- ζ in period $t + 1$, plus
- the right to sell the claim for p_{t+1} next period

The price satisfies (6.44) with $d_t = \zeta$, or

$$p_t = \mathbb{E}_t \left[m_{t+1} (\zeta + p_{t+1}) \right]$$

We maintain the stochastic discount factor (6.57), so this becomes

$$p_t = \mathbb{E}_t \left[\beta g_{t+1}^{-\gamma} (\zeta + p_{t+1}) \right] \tag{6.59}$$

Guessing a solution of the form $p_t = p(X_t)$ and conditioning on $X_t = x$, we get

$$p(x) = \beta \sum_{y \in S} g(y)^{-\gamma} (\zeta + p(y)) P(x, y)$$

Letting $M(x, y) = P(x, y) g(y)^{-\gamma}$ and rewriting in vector notation yields the solution

$$p = (I - \beta M)^{-1} \beta M \zeta \mathbb{1} \tag{6.60}$$

Pricing an Option to Purchase the Consol

Let's now price options of varying maturity that give the right to purchase a consol at a price p_S

An infinite horizon call option

We want to price an infinite horizon option to purchase a consol at a price p_S

The option entitles the owner at the beginning of a period either to

1. purchase the bond at price p_S now, or
2. Not to exercise the option now but to retain the right to exercise it later

Thus, the owner either *exercises* the option now, or chooses *not to exercise* and wait until next period

This is termed an infinite-horizon *call option* with *strike price* p_S

The owner of the option is entitled to purchase the consol at the price p_S at the beginning of any period, after the coupon has been paid to the previous owner of the bond

The fundamentals of the economy are identical with the one above, including the stochastic discount factor and the process for consumption

Let $w(X_t, p_S)$ be the value of the option when the time t growth state is known to be X_t but *before* the owner has decided whether or not to exercise the option at time t (i.e., today)

Recalling that $p(X_t)$ is the value of the consol when the initial growth state is X_t, the value of the option satisfies

$$w(X_t, p_S) = \max \left\{ \beta \, \mathbb{E}_t \frac{u'(c_{t+1})}{u'(c_t)} w(X_{t+1}, p_S), \; p(X_t) - p_S \right\}$$

The first term on the right is the value of waiting, while the second is the value of exercising now

We can also write this as

$$w(x, p_S) = \max \left\{ \beta \sum_{y \in S} P(x,y) g(y)^{-\gamma} w(y, p_S), \; p(x) - p_S \right\} \tag{6.61}$$

With $M(x,y) = P(x,y)g(y)^{-\gamma}$ and w as the vector of values $(w(x_i), p_S)_{i=1}^n$, we can express (6.61) as the nonlinear vector equation

$$w = \max\{\beta M w, \; p - p_S \mathbb{1}\} \tag{6.62}$$

To solve (6.62), form the operator T mapping vector w into vector Tw via

$$Tw = \max\{\beta M w, \; p - p_S \mathbb{1}\}$$

Start at some initial w and iterate to convergence with T

Here's a plot of w compared to the consol price when $P_S = 40$

The code with all details can be found here

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

$$\mathbb{E}_t m_{t+1} = \beta \sum_{y \in S} P(x, y) g(y)^{-\gamma}$$

We can write this as

$$m_1 = \beta M \mathbb{1}$$

where the i-th element of m_1 is the reciprocal of the one-period gross risk-free interest rate in state x_i

Other terms

Let m_j be an $n \times 1$ vector whose i th component is the reciprocal of the j -period gross risk-free interest rate in state x_i

Then $m_1 = \beta M$, and $m_{j+1} = M m_j$ for $j \geq 1$

Implementation

The file asset_pricing.py provides some functions for computing prices of the Lucas tree, consol and call option described above

Its contents are as follows

```
"""
Filename: asset_pricing.py

Computes asset prices with a Lucas style discount factor when the endowment
obeys geometric growth driven by a finite state Markov chain.  That is,

.. math::
    d_{t+1} = g(X_{t+1}) d_t

where

    * :math:`\{X_t\}` is a finite Markov chain with transition matrix P.

    * :math:`g` is a given positive-valued function

References
----------

    http://quant-econ.net/py/markov_asset.html

"""
import numpy as np
import quantecon as qe
from numpy.linalg import solve, eigvals


class AssetPriceModel:
    r"""
    A class that stores the primitives of the asset pricing model.

    Parameters
    ----------
    beta : scalar, float
        Discount factor
    mc : MarkovChain
        Contains the transition matrix and set of state values for the state
        process
    gamma : scalar(float)
```

```
        Coefficient of risk aversion
    g : callable
        The function mapping states to growth rates

    """
    def __init__(self, beta=0.96, mc=None, gamma=2.0, g=np.exp):
        self.beta, self.gamma = beta, gamma
        self.g = g

        # == A default process for the Markov chain == #
        if mc is None:
            self.rho = 0.9
            self.sigma = 0.02
            self.mc = qe.tauchen(self.rho, self.sigma, 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.beta:
            msg = "Spectral radius condition failed with radius = %f" % sr
            raise ValueError(msg)



def tree_price(ap):
    """
    Computes the price-dividend ratio of the Lucas tree.

    Parameters
    ----------
    ap: AssetPriceModel
        An instance of AssetPriceModel containing primitives

    Returns
    -------
    v : array_like(float)
        Lucas tree price-dividend ratio

    """
    # == Simplify names, set up matrices  == #
    beta, gamma, P, y = ap.beta, ap.gamma, ap.mc.P, ap.mc.state_values
    J = P * ap.g(y)**(1 - gamma)

    # == 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 - beta * J, beta * J @ Ones)

    return v


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

    """
    # == 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 solution exists == #
    ap.test_stability(M)

    # == Compute price == #
    I = np.identity(ap.n)
    Ones = np.ones(ap.n)
    p = solve(I - beta * M, beta * zeta * M @ Ones)

    return p


def call_option(ap, zeta, p_s, epsilon=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

    epsilon : scalar(float), optional(default=1e-8)
        Tolerance for infinite horizon problem

    Returns
    -------
    w : array_like(float)
        Infinite horizon call option prices

    """
    # == Simplify names, set up matrices  == #
    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)
    w = np.zeros(ap.n)
    error = epsilon + 1
    while error > epsilon:
        # == Maximize across columns == #
        w_new = np.maximum(beta * 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
```

Exercise 1 asks you to make use of this code

Exercises

Exercise 1

Consider the following primitives

```
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
gamma = 2.0
beta = 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 2

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 (6.61)

A k period option expires after k periods

If we view today as date zero, a k period option gives the owner the right to exercise the option to purchase the risk-free consol at the strike price p_S at dates $0, 1, \ldots, k-1$

The option expires at time k

Thus, for $k = 1, 2, \ldots$, let $w(x, k)$ be the value of a k-period option

It obeys

$$w(x, k) = \max \left\{ \beta \sum_{y \in S} P(x, y) g(y)^{-\gamma} w(y, k-1), \; p(x) - p_S \right\}$$

where $w(x, 0) = 0$ for all x

We can express the preceding as the sequence of nonlinear vector equations

$$w_k = \max\{\beta M w_{k-1}, \; p - p_S \mathbb{1}\} \quad k = 1, 2, \ldots \quad \text{with } w_0 = 0$$

Write a function that computes w_k for any given k

Compute the value of the option with `k = 5` and `k=25` using parameter values as in Exercise 1

Is one higher than the other? Can you give intuition?

Solutions

Exercise 1

First let's enter the parameters:

```
n = 5
P = 0.0125 * np.ones((n, n))
P += np.diag(0.95 - 0.0125 * np.ones(5))
s = np.array([0.95, 0.975, 1.0, 1.025, 1.05])  # state values
mc = qe.MarkovChain(P, state_values=s)

gamma = 2.0
beta = 0.94
zeta = 1.0
p_s = 150.0
```

Next we'll create an instance of `AssetPriceModel` to feed into the functions.

```
apm = AssetPriceModel(beta=beta, mc=mc, gamma=gamma, g=lambda x: x)
```

Now we just need to call the relevent functions on the data:

```
tree_price(apm)
```

```
array([ 29.47401578,  21.93570661,  17.57142236,  14.72515002,  12.72221763])
```

```
consol_price(apm, zeta)
```

```
array([ 753.87100476,  242.55144082,  148.67554548,  109.25108965,
         87.56860139])
```

```
call_option(apm, zeta, p_s)
```

```
array([ 603.87100476,  176.8393343 ,  108.67734499,   80.05179254,
         64.30843748])
```

Let's show the last two functions as a plot.

```
import matplotlib.pyplot as plt

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 2

Here's a suitable function:

```
def finite_horizon_call_option(ap, zeta, p_s, k):
    """
    Computes k period option value.
    """
    # == 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 solution exists == #
    ap.test_stability(M)


    # == Compute option price == #
    p = consol_price(ap, zeta)
    w = np.zeros(ap.n)
    for i in range(k):
        # == Maximize across columns == #
        w = np.maximum(beta * M @ w, p - p_s)

    return w
```

Now let's compute the option values at `k=5` and `k=25`

```
fig, ax = plt.subplots()
for k in [5, 25]:
    w = finite_horizon_call_option(apm, zeta, p_s, k)
    ax.plot(s, w, label=r'$k = {}$'.format(k))
ax.legend()
plt.show()
```

Not surprisingly, the option has greater value with larger k. This is because the owner has a longer time horizon over which he or she may exercise the option.

The Lucas Asset Pricing Model

Contents

- *The Lucas Asset Pricing Model*
  - *Overview*
  - *The Lucas Model*
  - *Exercises*
  - *Solutions*

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

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 ϕ and $y_t \geq 0$

An asset is a claim on all or part of this endowment stream

The consumption goods $\{y_t\}_{t=0}^{\infty}$ are nonstorable, so holding assets is the only way to transfer wealth into the future

For the purposes of intuition, it's common to think of the productive unit as a "tree" that produces fruit

Based on this idea, a "Lucas tree" is a claim on the consumption endowment

Consumers

A representative consumer ranks consumption streams $\{c_t\}$ according to the time separable utility functional

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{6.63}$$

Here

- $\beta \in (0, 1)$ is a fixed discount factor
- u is a strictly increasing, strictly concave, continuously differentiable period utility function
- $\mathbb{E}$ is a mathematical expectation

Pricing a Lucas Tree

What is an appropriate price for a claim on the consumption endowment?

We'll price an *ex dividend* claim, meaning that

- the seller retains this period's dividend
- the buyer pays p_t today to purchase a claim on
  - y_{t+1} and
  - the right to sell the claim tomorrow at price p_{t+1}

Since this is a competitive model, the first step is to pin down consumer behavior, taking prices as given

Next we'll impose equilibrium constraints and try to back out prices

In the consumer problem, the consumer's control variable is the share π_t of the claim held in each period

Thus, the consumer problem is to maximize (6.63) subject to

$$c_t + \pi_{t+1} p_t \leq \pi_t y_t + \pi_t p_t$$

along with $c_t \geq 0$ and $0 \leq \pi_t \leq 1$ at each t

The decision to hold share π_t is actually made at time $t - 1$

But this value is inherited as a state variable at time t, which explains the choice of subscript

The dynamic program

We can write the consumer problem as a dynamic programming problem

Our first observation is that prices depend on current information, and current information is really just the endowment process up until the current period

In fact the endowment process is Markovian, so that the only relevant information is the current state $y \in \mathbb{R}_+$ (dropping the time subscript)

This leads us to guess an equilibrium where price is a function p of y

Remarks on the solution method

- Since this is a competitive (read: price taking) model, the consumer will take this function p as given
- In this way we determine consumer behavior given p and then use equilibrium conditions to recover p
- This is the standard way to solve competitive equilibrum models

Using the assumption that price is a given function p of y, we write the value function and constraint as

$$v(\pi, y) = \max_{c, \pi'} \left\{ u(c) + \beta \int v(\pi', G(y, z))\phi(dz) \right\}$$

subject to

$$c + \pi' p(y) \leq \pi y + \pi p(y) \tag{6.64}$$

We can invoke the fact that utility is increasing to claim equality in (6.64) and hence eliminate the constraint, obtaining

$$v(\pi, y) = \max_{\pi'} \left\{ u[\pi(y + p(y)) - \pi' p(y)] + \beta \int v(\pi', G(y, z))\phi(dz) \right\} \tag{6.65}$$

The solution to this dynamic programming problem is an optimal policy expressing either π' or c as a function of the state (π, y)

- Each one determines the other, since $c(\pi, y) = \pi(y + p(y)) - \pi'(\pi, y)p(y)$

Next steps

What we need to do now is determine equilibrium prices

It seems that to obtain these, we will have to

1. Solve this two dimensional dynamic programming problem for the optimal policy
2. Impose equilibrium constraints
3. Solve out for the price function $p(y)$ directly

However, as Lucas showed, there is a related but more straightforward way to do this

Equilibrium constraints

Since the consumption good is not storable, in equilibrium we must have $c_t = y_t$ for all t

In addition, since there is one representative consumer (alternatively, since all consumers are identical), there should be no trade in equilibrium

In particular, the representative consumer owns the whole tree in every period, so $\pi_t = 1$ for all t

Prices must adjust to satisfy these two constraints

The equilibrium price function

Now observe that the first order condition for (6.65) 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 (6.65) with respect to π, yielding

$$v_1'(\pi, y) = u'(c)(y + p(y))$$

Next we impose the equilibrium constraints while combining the last two equations to get

$$p(y) = \beta \int \frac{u'[G(y,z)]}{u'(y)} [G(y,z) + p(G(y,z))] \phi(dz) \tag{6.66}$$

In sequential rather than functional notation, we can also write this as

$$p_t = \mathbb{E}_t \left[\beta \frac{u'(c_{t+1})}{u'(c_t)} (c_{t+1} + p_{t+1}) \right] \tag{6.67}$$

This is the famous consumption-based asset pricing equation

Before discussing it further we want to solve out for prices

Solving the Model

Equation (6.66) is a *functional equation* in the unknown function p

The solution is an equilibrium price function p^*

Let's look at how to obtain it

Setting up the problem

Instead of solving for it directly we'll follow Lucas' indirect approach, first setting

$$f(y) := u'(y) p(y) \tag{6.68}$$

so that (6.66) becomes

$$f(y) = h(y) + \beta \int f[G(y,z)] \phi(dz) \tag{6.69}$$

Here $h(y) := \beta \int u'[G(y,z)] G(y,z) \phi(dz)$ is a function that depends only on the primitives

Equation (6.69) is a functional equation in f

The plan is to solve out for f and convert back to p via (6.68)

To solve (6.69) we'll use a standard method: convert it to a fixed point problem

First we introduce the operator T mapping f into Tf as defined by

$$(Tf)(y) = h(y) + \beta \int f[G(y,z)] \phi(dz) \tag{6.70}$$

The reason we do this is that a solution to (6.69) now corresponds to a function f^* satisfying $(Tf^*)(y) = f^*(y)$ for all y

In other words, a solution is a *fixed point* of T

This means that we can use fixed point theory to obtain and compute the solution

A little fixed point theory

Let $cb\mathbb{R}_+$ be the set of continuous bounded functions $f\colon \mathbb{R}_+ \to \mathbb{R}_+$

We now show that

1. T has exactly one fixed point f^* in $cb\mathbb{R}_+$
2. For any $f \in cb\mathbb{R}_+$, the sequence $T^k f$ converges uniformly to f^*

(Note: If you find the mathematics heavy going you can take 1–2 as given and skip to the *next section*)

Recall the Banach contraction mapping theorem

It tells us that the previous statements will be true if we can find an $\alpha < 1$ such that

$$\|Tf - Tg\| \leq \alpha \|f - g\|, \qquad \forall\, f, g \in cb\mathbb{R}_+ \tag{6.71}$$

Here $\|h\| := \sup_{x \in \mathbb{R}_+} |h(x)|$

To see that (6.71) is valid, pick any $f, g \in cb\mathbb{R}_+$ and any $y \in \mathbb{R}_+$

Observe that, since integrals get larger when absolute values are moved to the inside,

$$\begin{aligned}
|Tf(y) - Tg(y)| &= \left| \beta \int f[G(y,z)]\phi(dz) - \beta \int g[G(y,z)]\phi(dz) \right| \\
&\leq \beta \int |f[G(y,z)] - g[G(y,z)]|\, \phi(dz) \\
&\leq \beta \int \|f - g\| \phi(dz) \\
&= \beta \|f - g\|
\end{aligned}$$

Since the right hand side is an upper bound, taking the sup over all y on the left hand side gives (6.71) 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 $u(c) = c^{1-\gamma}/(1-\gamma)$, 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

```
r"""
Filename: lucastree.py
```

```
Reference: http://quant-econ.net/py/lucas_model.html

Solves the price function for the Lucas tree in a continuous state
setting, using piecewise linear approximation for the sequence of
candidate price functions.  The consumption endownment follows the log
linear AR(1) process

.. math::

    log y' = \alpha log y + \sigma \epsilon

where y' is a next period y and epsilon is an iid standard normal shock.
Hence

.. math::

    y' = y^{\alpha} * \xi,

where

.. math::

    \xi = e^(\sigma * \epsilon)

The distribution phi of xi is

.. math::

    \phi = LN(0, \sigma^2),

where LN means lognormal.

"""
import numpy as np
from scipy.stats import lognorm
from scipy.integrate import fixed_quad
from quantecon import compute_fixed_point


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)
        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
    phi : scipy.stats.lognorm
        The distribution for the shock process

    Examples
    --------
    >>> tree = LucasTree(gamma=2, beta=0.95, alpha=0.90, sigma=0.1)
    >>> price_vals = compute_lt_price(tree)

    """

    def __init__(self,
            gamma=2,
            beta=0.95,
            alpha=0.90,
            sigma=0.1,
            grid_size=100):

        self.gamma = gamma
        self.beta = beta
        self.alpha = alpha
        self.sigma = 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.phi = lognorm(sigma)
        self.draws = self.phi.rvs(500)

        # == h(y) = beta * int G(y,z)^(1-gamma) phi(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.

    """
    grid,  h = tree.grid, tree.h
    alpha, beta = tree.alpha, tree.beta
    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] + beta * np.mean(Af(y**alpha * z_vec))

    return Tf


def compute_lt_price(tree, error_tol=1e-6, max_iter=500, verbose=0):
    """
    Compute the equilibrium price function associated with Lucas
    tree lt
```

```
    Parameters
    ----------
    tree : An instance of LucasTree
        Contains parameters

    error_tol, max_iter, verbose
        Arguments to be passed directly to
        `quantecon.compute_fixed_point`. See that docstring for more
        information

    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
    gamma = tree.gamma

    # == Create storage array for compute_fixed_point. Reduces  memory
    # allocation and speeds code up == #
    Tf = np.empty(grid_size)

    # == Initial guess, just a vector of zeros == #
    f_init = np.zeros(grid_size)
    f = compute_fixed_point(lucas_operator,
                f_init,
                error_tol,
                max_iter,
                verbose,
                10,
                'iteration',
                tree,
                Tf=Tf)

    price = f * grid**gamma

    return price
```

An example of usage is given in the docstring and repeated here

```
tree = LucasTree()
price_vals = compute_lt_price(tree)
```

Here's the resulting price function

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 blue 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

Exercises

Exercise 1

Replicate *the figure* to show how discount rates affect prices

Solutions

```
import matplotlib.pyplot as plt
```

Exercise 1

Note that this code assumes you have run the lucastree.py script embedded above

```
fig, ax = plt.subplots(figsize=(10,7))

ax.set_xlabel(r'$y$', fontsize=16)
ax.set_ylabel(r'price', fontsize=16)

for beta in (.95, 0.98):
    print("Computing at beta = {}".format(beta))
    tree = LucasTree(beta=beta)
    grid = tree.grid
    price_vals = compute_lt_price(tree)
    label = r'$\beta = {}$'.format(beta)
    ax.plot(grid, price_vals, lw=2, alpha=0.7, label=label)

ax.legend(loc='upper left')
ax.set_xlim(min(grid), max(grid))
plt.show()
```

```
Computing at beta = 0.95
Computing at beta = 0.98
```

A Harrison-Kreps Model of Asset Prices

Contents

- *A Harrison-Kreps Model of Asset Prices*
  - *Overview*
  - *Structure of the Model*
  - *Solving the Model*

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*

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 $S \in \{1, 2\}$

The dividend obeys

$$d_t = \begin{cases} 0 & \text{if } s_t = 1 \\ 1 & \text{if } s_t = 2 \end{cases}$$

The owner of a share at the beginning of time t is entitled to the dividend paid at time t

The owner of the share at the beginning of time t is also entitled to sell the share to another investor during time t

Two types $h = a, b$ of investors differ only in their beliefs about a Markov transition matrix P with typical element

$$P(i, j) = \mathbb{P}\{s_{t+1} = j \mid s_t = i\}$$

Investors of type a believe the transition matrix

$$P_a = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

Investors of type b think the transition matrix is

$$P_b = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

The stationary (i.e., invariant) distributions of these two matrices can be calculated as follows:

```
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
```

```
array([[ 0.57142857,  0.42857143]])
```

```
mcB.stationary_distributions
```

```
array([[ 0.42857143,  0.57142857]])
```

The stationary distribution of P_a is approximately $\pi_B = \begin{bmatrix} .57 & .43 \end{bmatrix}$

The stationary distribution of P_b is approximately $\pi_B = \begin{bmatrix} .43 & .57 \end{bmatrix}$

Ownership Rights

An owner of the asset at the end of time t is entitled to the dividend at time $t+1$ and also has the right to sell the asset at time $t+1$

Both types of investors are risk-neutral and both have the same fixed discount factor $\beta \in (0,1)$

In our numerical example, we'll set $\beta = .75$, just as Harrison and Kreps did

We'll eventually study the consequences of two different assumptions about the number of shares A relative to the resources that our two types of investors can invest in the stock

1. Both types of investors have enough resources (either wealth or the capacity to borrow) so that they can purchase the entire available stock of the asset[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 2 is the high dividend state

- In state 1, a type a agent is more optimistic about next period's dividend than a type b agent
- In state 2, a type b agent is more optimistic about next period's dividend

However, the stationary distributions $\pi_A = \begin{bmatrix} .57 & .43 \end{bmatrix}$ and $\pi_B = \begin{bmatrix} .43 & .57 \end{bmatrix}$ tell us that a type B person is more optimistic about the dividend process in the long run than is a type A person

[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 1 to 2 or from 2 to 1.

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

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 Kreps's 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

It records implications of Harrison and Kreps's specifications of P_a, P_b, β

s_t	1	2
p_a	1.33	1.22
p_b	1.45	1.91
$\bar{p}_a$	1.85	2.08
$\hat{p}_a$	1.85	1.69
$\hat{p}_b$	1.69	2.08
$\check{p}$	1	1

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
- $\bar{p}_a$ is the equilibrium price function under heterogeneous beliefs with optimistic 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
- $\check{p}$ is the equilibrium price function under heterogeneous beliefs with pessimistic marginal investors

We'll explain these values and how they are calculated one row at a time

Single Belief Prices

We'll start by pricing the asset under homogeneous beliefs

(This is the case treated in *the lecture* on asset pricing with finite Markov states)

Suppose that there is only one type of investor, either of type a or b, and that this investor always "prices the asset"

Let $p_h = \begin{bmatrix} p_h(1) \\ p_h(2) \end{bmatrix}$ be the equilibrium price vector when all investors are of type h

The price today equals the expected discounted value of tomorrow's dividend and tomorrow's price of the asset:

$$p_h(s) = \beta \left(P_h(s,1) p_i(1) + P_h(s,2)(1 + p_h(2)) \right), \quad s = 1, 2$$

These equations imply that the equilibrium price vector is

$$\begin{bmatrix} p_h(1) \\ p_h(2) \end{bmatrix} = \beta [I - \beta P_h]^{-1} P_h \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{6.72}$$

The first two rows of of the table report $p_a(s)$ and $p_b(s)$

Here's a function that can be used to compute these values

```
"""
Provides a function to solve for asset prices under one set of beliefs in the
Harrison -- Kreps model.

Authors: Chase Coleman, Tom Sargent
"""
import numpy as np
import scipy.linalg as la

def price_singlebeliefs(transition, dividend_payoff, beta=.75):
    """
    Function to Solve Single Beliefs
    """
    # First compute inverse piece
    imbq_inv = la.inv(np.eye(transition.shape[0]) - beta*transition)

    # Next compute prices
    prices = beta * np.dot(np.dot(imbq_inv, transition), dividend_payoff)

    return prices
```

The code shown above can be downloaded here

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's key equation:

$$\bar{p}(s) = \beta \max \left\{ P_a(s,1)\bar{p}(1) + P_a(s,2)(1+\bar{p}(2)),\ P_b(s,1)\bar{p}(1) + P_b(s,2)(1+\bar{p}(2)) \right\} \tag{6.73}$$

for $s = 1, 2$

The marginal investor who prices the asset in state s is of type a if

$$P_a(s,1)\bar{p}(1) + P_a(s,2)(1+\bar{p}(2)) > P_b(s,1)\bar{p}(1) + P_b(s,2)(1+\bar{p}(2))$$

The marginal investor is of type b if

$$P_a(s,1)\bar{p}(1) + P_a(s,2)(1+\bar{p}(2)) < P_b(s,1)\bar{p}(1) + P_b(s,2)(1+\bar{p}(2))$$

Thus the marginal investor is the (temporarily) optimistic type.

Equation (6.73) 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 (6.73), namely

$$\bar{p}^{j+1}(s) = \beta \max \left\{ P_a(s,1)\bar{p}^j(1) + P_a(s,2)(1+\bar{p}^j(2)),\ P_b(s,1)\bar{p}^j(1) + P_b(s,2)(1+\bar{p}^j(2)) \right\} \tag{6.74}$$

for $s = 1, 2$

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

$$\hat{p}_a(s) = \begin{cases} \bar{p}(1) & \text{if } s_t = 1 \\ \beta(P_a(2,1)\bar{p}(1) + P_a(2,2)(1+\bar{p}(2))) & \text{if } s_t = 2 \end{cases}$$

- Investors of type b are willing to pay the following price for the asset

$$\hat{p}_b(s) = \begin{cases} \beta(P_b(1,1)\bar{p}(1) + P_b(1,2)(1+\bar{p}(2))) & \text{if } s_t = 1 \\ \bar{p}(2) & \text{if } s_t = 2 \end{cases}$$

Evidently, $\hat{p}_a(2) < \bar{p}(2)$ and $\hat{p}_b(1) < \bar{p}(1)$

Investors of type a want to sell the asset in state 2 while investors of type b want to sell it in state 1

- The asset changes hands whenever the state changes from 1 to 2 or from 2 to 1
- The valuations $\hat{p}_a(s)$ and $\hat{p}_b(s)$ are displayed in the fourth and fifth rows of the table
- Even the pessimistic investors who don't buy the asset think that it is worth more than they think future dividends are worth

Here's code to solve for $\bar{p}$, $\hat{p}_a$ and $\hat{p}_b$ using the iterative method described above

```
"""
Provides a function to solve for asset prices under optimistic beliefs in the
Harrison -- Kreps model.

Authors: Chase Coleman, Tom Sargent
"""
import numpy as np

def price_optimisticbeliefs(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]
    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.max([np.dot(q, p_old) + np.dot(q, dividend_payoff)
                               for q in transitions], 1)

        # If we succed in converging, break out of for loop
        if np.max(np.sqrt((p_new - p_old)**2)) < 1e-12:
            break
```

```
    ptwiddle = beta * np.min([np.dot(q, p_old) + np.dot(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
```

The code shown above can be downloaded here

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 (6.73), the equilibrium price satisfies

$$\check{p}(s) = \beta \min \left\{ P_a(s,1)\check{p}(1) + P_a(s,2)(1+\check{p}(2)),\ P_b(s,1)\check{p}(1) + P_b(s,2)(1+\check{p}(2)) \right\} \tag{6.75}$$

and the marginal investor who prices the asset is always the one that values it *less* highly than does the other type

Now the marginal investor is always the (temporarily) pessimistic type

Notice from the sixth row of that the pessimistic price $\underline{p}$ is lower than the homogeneous belief prices p_a and p_b in both states

When pessimistic investors price the asset according to (6.75), optimistic investors think that the asset is underpriced

If they could, optimistic investors would willingly borrow at the one-period gross interest rate β^{-1} to purchase more of the asset

Implicit constraints on leverage prohibit them from doing so

When optimistic investors price the asset as in equation (6.73), pessimistic investors think that the asset is overpriced and would like to sell the asset short

Constraints on short sales prevent that

Here's code to solve for $\check{p}$ using iteration

```
"""
Provides a function to solve for asset prices under pessimistic beliefs in the
Harrison -- Kreps model.

Authors: Chase Coleman, Tom Sargent
"""
import numpy as np

def price_pessimisticbeliefs(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([np.dot(q, p_old) + np.dot(q, dividend_payoff)
                               for q in transitions], 1)

        # If we succed in converging, break out of for loop
        if np.max(np.sqrt((p_new - p_old)**2)) < 1e-12:
            break

    return p_new
```

The code shown above can be downloaded here

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 asset's underlying dividend stream

Scheinkman stresses these features of the Harrison-Kreps model:

- Compared to the homogeneous beliefs setting leading to the pricing formula, high volume occurs when the Harrison-Kreps pricing formula prevails

Type a investors sell the entire stock of the asset to type b investors every time the state switches from $s_t = 1$ to $s_t = 2$

Type b investors sell the asset to type a investors every time the state switches from $s_t = 2$ to $s_t = 1$

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

The Aiyagari Model

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 *[LS12]*

A continuous time version of the model by SeHyoun Ahn and Benjamin Moll can be found here

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

$$a_{t+1} + c_t \leq w z_t + (1+r) a_t \quad c_t \geq 0, \quad \text{and} \quad a_t \geq -B$$

where

- c_t is current consumption

- a_t is assets
- z_t is an exogenous component of labor income capturing stochastic unemployment risk, etc.
- w is a wage rate
- r is a net interest rate
- B is the maximum amount that the agent is allowed to borrow

The exogenous process $\{z_t\}$ follows a finite state Markov chain with given stochastic matrix P

The wage and interest rate are fixed over time

In this simple version of the model, households supply labor inelastically because they do not value leisure

Firms

Firms produce output by hiring capital and labor

Firms act competitively and face constant returns to scale

Since returns to scale are constant the number of firms does not matter

Hence we can consider a single (but nonetheless competitive) representative firm

The firm's output is

$$Y_t = AK_t^\alpha N^{1-\alpha}$$

where

- A and α are parameters with $A > 0$ and $\alpha \in (0, 1)$
- K_t is aggregate capital
- N is total labor supply (which is constant in this simple version of the model)

The firm's problem is

$$max_{K,N} \left\{ AK_t^\alpha N^{1-\alpha} - (r+\delta)K - wN \right\}$$

The parameter δ is the depreciation rate

From the first-order condition with respect to capital, the firm's inverse demand for capital is

$$r = A\alpha \left(\frac{N}{K}\right)^{1-\alpha} - \delta \tag{6.76}$$

Using this expression and the firm's first-order condition for labor, we can pin down the equilibrium wage rate as a function of r as

$$w(r) = A(1-\alpha)(A\alpha/(r+\delta))^{\alpha/(1-\alpha)} \tag{6.77}$$

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 (6.76) and a wage rate $w(r)$ as given in (6.77)
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

Code

Let's look at how we might compute such an equilibrium in practice

To solve the household's dynamic programming problem we'll use the DiscreteDP class from QuantEcon.py

Our first task is the least exciting one: write code that maps parameters for a household problem into the `R` and `Q` matrices needed to generate an instance of `DiscreteDP`

Below is a piece of boilerplate code that does just this

It comes from the file aiyagari_household.py

In reading the code, the following information will be helpful

- `R` needs to be a matrix where `R[s, a]` is the reward at state `s` under action `a`
- `Q` needs to be a three dimensional array where `Q[s, a, s']` is the probability of transitioning to state `s'` when the current state is `s` and the current action is `a`

(For a detailed discussion of `DiscreteDP` see *this lecture*)

Here we take the state to be $s_t := (a_t, z_t)$, where a_t is assets and z_t is the shock

The action is the choice of next period asset level a_{t+1}

We use Numba to speed up the loops so we can update the matrices efficiently when the parameters change

The class also includes a default set of parameters that we'll adopt unless otherwise specified

```
"""
Created on Wed Sep 23 17:00:17 EDT 2015
@authors: John Stachurski, Thomas Sargent

"""

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, ..., n}.  To this end, (a_i, z_i) index pairs are mapped to s_i
    indices according to the rule

        s_i = a_i * z_size + z_i

    To invert this map, use

        a_i = s_i // z_size  (integer division)
        z_i = s_i % z_size

    """


    def __init__(self,
                 r=0.01,       # interest rate
                 w=1.0,        # wages
                 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)

        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]


@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
```

In the following examples our import statements assume that this code is stored as `aiyagari_household.py` in the present working directory

(Or you can copy it into a Jupyter notebook cell and delete the corresponding import statement)

As a first example of what we can do, let's compute and plot an optimal accumulation policy at fixed prices

```
"""
Created on Wed Sep 23 17:00:17 EDT 2015
@authors: John Stachurski, Thomas Sargent

Filename: aiyagari_compute_policy.py

Computes and plots the optimal policy of a household from the Aiyagari model,
given prices.

"""
import numpy as np
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.beta)

# 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 = r'$z = {}$'.format(z_vals[i], '.2f')
    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

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

```
"""
Created on Wed Sep 23 17:00:17 EDT 2015
@authors: John Stachurski, Thomas Sargent
"""

import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
from numba import jit
from quantecon.markov import DiscreteDP


A = 1.0
N = 1.0
alpha = 0.33
beta = 0.96
delta = 0.05


def r_to_w(r):
    """
    Equilibrium wages associated with a given interest rate r.
    """
    return A * (1 - alpha) * (A * alpha / (r + delta))**(alpha / (1 - alpha))

def rd(K):
    """
    Inverse demand curve for capital.  The interest rate associated with a
    given demand for capital K.
    """
    return A * alpha * (N / K)**(1 - alpha) - delta


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, beta)
    # 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.beta)

# Create a grid of r values at which to compute demand and supply of capital
num_points = 20
r_vals = np.linspace(0.005, 0.04, num_points)

# Compute supply of capital
k_vals = np.empty(num_points)
for i, r in enumerate(r_vals):
    k_vals[i] = prices_to_capital_stock(am, r)

# Plot against demand for capital by firms
fig, ax = plt.subplots(figsize=(11, 8))
ax.plot(k_vals, r_vals, lw=2, alpha=0.6, label='supply of capital')
ax.plot(k_vals, rd(k_vals), lw=2, alpha=0.6, label='demand for capital')
ax.grid()
ax.set_xlabel('capital')
ax.set_ylabel('interest rate')
ax.legend(loc='upper right')

plt.show()
```

Here's the corresponding plot

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)*
  - *Model 2 (One-Period Risk Free Debt Only)*
  - *Example: Tax Smoothing with Complete Markets*
  - *Linear state space version of complete markets model*

Overview

This lecture describes two types of consumption-smoothing model

- 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 β^{-1}, where $\beta \in (0, 1)$ is also a consumer's intertemporal discount factor

Barro *[Bar79]* made an analogous assumption about the risk-free interest rate in a tax-smoothing model that we regard as isomorphic to Hall's consumption-smoothing model

We maintain Hall and Barro's assumption about the interest rate when we describe an incomplete markets version of our model

In addition, we extend their assumption about the interest rate to an appropriate counterpart 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 consumer's debt as the government's *assets*

We'll consider two closely related assumptions about the consumer's exogenous non-financial income process (or in the tax-smoothing interpretation, the government's 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

We'll 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 *The Permanent Income Model: II* 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

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 period's 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, there are N such securities 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 β^{-1}

Finite state Markov income process

In each version of the consumption-smoothing model, nonfinancial income is governed by a two-state Markov chain (it's easy to generalize this to an N state chain)

In particular, the *state of the world* is given by s_t that follows a Markov chain with transition probability matrix

$$P_{ij} = \mathbb{P}\{s_{t+1} = \bar{s}_j \,|\, s_t = \bar{s}_i\}$$

Nonfinancial income $\{y_t\}$ obeys

$$y_t = \begin{cases} \bar{y}_1 & \text{if } s_t = \bar{s}_1 \\ \bar{y}_2 & \text{if } s_t = \bar{s}_2 \end{cases}$$

A consumer wishes to maximize

$$\mathbb{E}\left[\sum_{t=0}^{\infty}\beta^t u(c_t)\right] \quad \text{where} \quad u(c_t) = -(c_t-\gamma)^2 \quad \text{and} \quad 0<\beta<1 \tag{6.78}$$

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

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

Model 1 (Complete Markets)

At each date $t \geq 0$, the consumer trades **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)

Exogenous means that the decision maker can take them as given and unaffected by his decisions

In Markov state s_t at time t, one unit of consumption in state s_{t+1} at time $t+1$ costs $q(s_{t+1} \mid s_t)$ units of the time t consumption good

At time $t = 0$, the consumer starts with an inherited level of debt due at time 0 of b_0 units of time 0 consumption goods

The consumer's budget constraint at $t \geq 0$ in Markov state s_t is

$$c_t + b_t \leq y(s_t) + \sum_j q(\bar{s}_j \mid s_t)\, b_{t+1}(\bar{s}_j \mid s_t)$$

where $b_{t+1}(\bar{s}_j \mid s_t)$ are the consumer's time t sales of the time $t+1$ consumption good in Markov state $\bar{s}_j$

An analogue of Hall's assumption that the one-period risk-free gross interest rate is β^{-1} is

$$q(\bar{s}_j \mid \bar{s}_i) = \beta P_{ij} \tag{6.79}$$

To understand this, observe that in state $\bar{s}_i$ it costs $\sum_j q(\bar{s}_j \mid \bar{s}_i)$ 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

$$\sum_j q(\bar{s}_j \mid \bar{s}_i) = \sum_j \beta P_{ij} = \beta$$

This confirms that (6.79) is a natural analogue of Hall's assumption about the risk-free one-period interest rate

First-order necessary conditions for maximizing the consumer's 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 (6.79) on Arrow security prices,

$$c_{t+1} = c_t \tag{6.80}$$

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

We'll make the plausible guess that

$$b_{t+1}(\bar{s}_j \mid s_t = \bar{s}_i) = b(\bar{s}_j), \quad i = 1, 2; \;\; j = 1, 2$$

so that the amount borrowed today turns out to depend only on *tomorrow's* Markov state. (Why is this is a plausible guess?)

To determine $\bar{c}$, we shall pursue the implications of our guess about the consumer's debt level choices and the consumer's budget constraints in each Markov state today

For $t \geq 1$, we write these as

$$\begin{aligned} \bar{c} + b(\bar{s}_1) &= y(\bar{s}_1) + q(\bar{s}_1 \mid \bar{s}_1) b(\bar{s}_1) + q(\bar{s}_2 \mid \bar{s}_1) b(\bar{s}_2) \\ \bar{c} + b(\bar{s}_2) &= y(\bar{s}_2) + q(\bar{s}_1 \mid \bar{s}_2) b(\bar{s}_1) + q(\bar{s}_2 \mid \bar{s}_2) b(\bar{s}_2), \end{aligned} \tag{6.81}$$

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

$$\bar{c} + b_0 = y(\bar{s}_1) + q(\bar{s}_1 \mid \bar{s}_1) b(\bar{s}_1) + q(\bar{s}_2 \mid \bar{s}_1) b(\bar{s}_2) \tag{6.82}$$

If we substitute (6.82) into the first equation of (6.81) and rearrange, we discover that

$$b(\bar{s}_1) = b_0 \tag{6.83}$$

We can then use the second equation of (6.81) to deduce the restriction

$$y(\bar{s}_1) - y(\bar{s}_2) + [q(\bar{s}_1 \mid \bar{s}_1) - q(\bar{s}_1 \mid \bar{s}_2) - 1] b_0 + [q(\bar{s}_2 \mid \bar{s}_1) + 1 - q(\bar{s}_2 \mid \bar{s}_2)] b(\bar{s}_2) = 0, \tag{6.84}$$

an equation in the unknown $b(\bar{s}_2)$

Knowing $b(\bar{s}_1)$ and $b(\bar{s}_2)$, we can solve equation (6.82) 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 consumer's debt neither accumulates, nor decumulates, nor drifts. Instead the debt level each period is an exact function of the Markov state, so in the two-state Markov case, it switches between two values

We 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

Code

Here's some code that, among other things, contains a function called *consumption_complete()*

This function computes $b(\bar{s}_1), b(\bar{s}_2), \bar{c}$ as outcomes given a set of parameters, under the assumption of complete markets

```
import numpy as np
import quantecon as qe
import scipy.linalg as la

## Parameters

class ConsumptionProblem:
    """
    The data for a consumption problem, including some default values.
    """

    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 * 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 * P        # assumed price system

    # Using equation (7) calculate b2
    b2 = (y2 - y1 - (Q[0, 0] - Q[1, 0] - 1) * b1)/(Q[0, 1] + 1 - Q[1, 1])

    # Using equation (5) calculate c_bar
    c_bar = 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 * 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) * v - y) / beta

    for i, s in enumerate(s_path):
        c_path[i] = (1 - beta) * (v - b_path[i] * np.ones((2, 1)))[s, 0]
        b_path[i + 1] = b_path[i] + 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

```
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 * 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 Gallatin (1807) *[Gal37]*)

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 of β^{-1}

Given an initial debt b_0 at time 0, the consumer faces a sequence of budget constraints

$$c_t + b_t = y_t + \beta b_{t+1}, \quad t \geq 0$$

where β is the price at time t of a risk-free claim on one unit of time consumption at time $t+1$

First-order conditions for the consumer's problem are

$$\sum_j u'(c_{t+1,j})P_{ij} = u'(c_{t,i})$$

For our assumed quadratic utility function this implies

$$\sum_j c_{t+1,j}P_{ij} = c_{t,i}, \tag{6.85}$$

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

$$b_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - (1-\beta)^{-1} c_t \tag{6.86}$$

and

$$c_t = (1-\beta)\left[\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - b_t\right] \tag{6.87}$$

which expresses c_t as a net interest rate factor $1-\beta$ times the sum of the expected present value of non-financial income $\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$ and financial wealth $-b_t$

Substituting (6.87) into the one-period budget constraint and rearranging leads to

$$b_{t+1} - b_t = \beta^{-1}\left[(1-\beta)\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - y_t\right] \tag{6.88}$$

Now let's do a useful calculation that will yield a convenient expression for the key term $\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$ in our Markov setting

Define

$$v_t := \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$$

In our Markov setting, $v_t = v(1)$ when $s_t = \bar{s}_1$ and $v_t = v(2)$ when $s_t = \bar{s}_2$

Therefore, we can write

$$\begin{aligned} v(1) &= y(1) + \beta P_{11} v(1) + \beta P_{12} v(2) \\ v(2) &= y(2) + \beta P_{21} v(1) + \beta P_{22} v(2) \end{aligned}$$

or

$$\vec{v} = \vec{y} + \beta P \vec{v}$$

where $\vec{v} = \begin{bmatrix} v(1) \\ v(2) \end{bmatrix}$ and $\vec{y} = \begin{bmatrix} y(1) \\ y(2) \end{bmatrix}$

We can also write the last expression as

$$\vec{v} = (I - \beta P)^{-1}\vec{y}$$

In our Markov setting, from expression (6.87), consumption at date t when debt is b_t and the Markov state today is $s_t = i$ is evidently

$$c(b_t, i) = (1-\beta)\left([(I-\beta P)^{-1}\vec{y}]_i - b_t\right) \tag{6.89}$$

and the increment in debt is

$$b_{t+1} - b_t = \beta^{-1}[(1-\beta)v(i) - y(i)] \tag{6.90}$$

Summary of outcomes

In contrast to outcomes in the complete markets model, in the incomplete markets model

- consumption drifts over time as a random walk; the level of consumption at time t depends on the level of debt that the consumer brings into the period as well as the expected discounted present value of nonfinancial income at t
- the consumer's debt drifts upward over time in response to low realizations of nonfinancial income and drifts downward over time in response to high realizations of nonfinancial income
- the drift over time in the consumer's debt and the dependence of current consumption on today's debt level account for the drift over time in consumption

Code for the incomplete markets model

The code above also contains a function called *consumption_incomplete()* that uses (6.89) and (6.90) 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 economies

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

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', fontsize = 17)
ax[0].plot(np.arange(N_simul), c_path, label = 'incomplete market', lw = 3)
ax[0].plot(np.arange(N_simul), c_bar * np.ones(N_simul), label = 'complete␣
↪market', lw = 3)
ax[0].plot(np.arange(N_simul), y_path, label = 'income', lw = 2, color = sb.
↪color_palette()[3], alpha = .6, linestyle = '--')
ax[0].legend(loc = 'best', fontsize = 15)
ax[0].set_xlabel('Periods', fontsize = 13)
ax[0].set_ylim([1.4, 2.1])

ax[1].set_title('Debt paths', fontsize = 17)
ax[1].plot(np.arange(N_simul), debt_path, label = 'incomplete market', lw = 2)
ax[1].plot(np.arange(N_simul), debt_complete[s_path], label = 'complete market
↪', lw = 2)
ax[1].plot(np.arange(N_simul), y_path, label = 'income', lw = 2, color = sb.
↪color_palette()[3],
           alpha = .6, linestyle = '--')
ax[1].legend(loc = 'best', fontsize = 15)
ax[1].axhline(0, color = 'k', lw = 1)
ax[1].set_xlabel('Periods', fontsize = 13)

plt.show()
```

In the graph on the left, for the same sample path of nonfinancial income y_t, notice that

- consumption is constant when there are complete markets, but it varies as a random walk in the incomplete markets version of the model
- the consumer's debt oscillates between two values that are functions of the Markov state in the complete markets model, while the consumer's debt drifts in a "unit root" fashion in the incomplete markets economy

Using the isomorphism

We can simply relabel variables to acquire tax-smoothing interpretations of our two models

The relabeled code can be found here

The resulting figure is below

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-expenditure war 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

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 $b_0 = 0$, so that $b_1 = 0$

Here's our Python code to compute a quantitative example with zero debt in peace time:

```
import numpy as np

# Parameters

beta = .96
y = [1, 2]
b0 = 0
P = np.asarray([[.8, .2],
                [.4, .6]])

cp = ConsumptionProblem(beta, y, b0, P)
Q = beta*P
N_simul = 150

c_bar, b1, b2 = consumption_complete(cp)
debt_complete = np.asarray([b1, b2])

print("P = ", P)
print("Q= ", Q, "\n")
print("Govt expenditures in peace and war =", y)
print("Constant tax collections = ", c_bar)
print("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] * b2
print("Spending on Arrow war security in peace = ", AS1)
AS2 = Q[1,1]*b2
print("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("T+b in peace = ",TB1 )
TB2 = c_bar + b2
print("T+b in war = ", TB2)


print("\n")
print("Total government spending in peace and war")
G1= y[0] + AS1
G2 = y[1] + AS2
print("total govt spending in peace = ", G1)
print("total govt spending in war = ", G2)


print("\n")
print("Let's see ex post and ex ante returns on Arrow securities")
```

```
Pi= np.reciprocal(Q)
exret= Pi
print("Ex post returns to purchase of Arrow securities = ", exret)
exant = Pi*P
print("Ex ante returns to purchase of Arrow securities ", exant)
```

Here's the output it produces

```
P =  [[ 0.8  0.2]
     [ 0.4  0.6]]
Q =  [[ 0.768  0.192]
     [ 0.384  0.576]]

Govt expenditures in peace and war = [1, 2]
Constant tax collections =  1.31168831169
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 =  0.311688311688
Spending on Arrow war security in war =  0.935064935065


Government tax collections plus asset levels in peace and war
T+b in peace =  1.31168831169
T+b in war =  2.93506493506


Total government spending in peace and war
total govt spending in peace =  1.31168831169
total govt spending in war =  2.93506493506


Let's see ex post and ex ante returns on Arrow securities
Ex post returns to purchase of Arrow securities =  [[ 1.30208333  5.20833333]
 [ 2.60416667  1.73611111]]
Ex ante returns to purchase of Arrow securities  [[ 1.04166667  1.04166667]
 [ 1.04166667  1.04166667]]
```

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 $b_0 = -10$, so that the government starts the war in debt

Linear state space version of complete markets model

Now we'll use a setting like that in *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 β^{-1}
- the consumer's 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*

We'll write down a complete markets counterpart of that model

So now we'll suppose that nonfinancial income is governed by the state space system

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= S_y x_t \end{aligned}$$

where x_t is an $n \times 1$ vector and $w_{t+1} \sim \mathcal{N}(0, I)$ is IID over time

Again, as a counterpart of the Hall-Barro assumption that the risk-free gross interest rate is β^{-1}, we assume the scaled prices of one-period ahead Arrow securities are

$$p_{t+1}(x_{t+1} \mid x_t) = \beta\phi(x_{t+1} \mid Ax_t, CC') \tag{6.91}$$

where $\phi(\cdot \mid \mu, \Sigma)$ is a multivariate Gaussian distribution with mean vector μ and covariance matrix Σ

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 defined in (6.91), 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 shall assume that $u(c_t) = -(c_t - \gamma)^2$, so that the above utility functional becomes

$$-\sum_{t=0}^{\infty} \beta^t (c_t - \gamma)^2, \quad 0 < \beta < 1$$

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 consumer's 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

$$b_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j (y_{t+j} - \bar{c})$$

or

$$b_t = S_y (I - \beta A)^{-1} x_t - \frac{1}{1-\beta} \bar{c} \tag{6.92}$$

where the value of $\bar{c}$ satisfies

$$b_0 = S_y (I - \beta A)^{-1} x_0 - \frac{1}{1-\beta} \bar{c} \tag{6.93}$$

where b_0 is an initial level of the consumer's 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 (6.93) and the consumer's debt is a fixed function of the state x_t described by (6.92)

Here's an example that shows how in this setting the availability of insurance against fluctuating nonfinancial income allows the consumer completely to smooth consumption across time and across states of the world.

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

N_simul = 150
```

```
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]) - beta*A)

    # 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


if __name__ == '__main__':

    # Define parameters
    alpha, rho1, rho2 = 10.0, 0.9, 0.0
    sigma = 1.0
    # N_simul = 1
    # T = N_simul
    A = np.array([[1., 0., 0.],
                  [alpha, rho1, rho2],
                  [0., 1., 0.]])
    C = np.array([[0.], [sigma], [0.]])
    S_y = np.array([[1,  1.0, 0.]])
    beta, b0 = 0.95, -10.0
    x0 = np.array([1.0, alpha/(1-rho1), alpha/(1-rho1)])

    # 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', lw = 3)
    ax[0].plot(np.arange(N_simul), y_hist_com, label = 'income',
               lw = 2, color = sb.color_palette()[3], alpha = .6, linestyle =
↪'--')
    ax[0].legend(loc = 'best', fontsize = 15)
    ax[0].set_xlabel('Periods', fontsize = 13)
    ax[0].set_ylim([-5.0, 110])

    # Debt plots
    ax[1].set_title('Debt and income', fontsize = 17)
    ax[1].plot(np.arange(N_simul), b_hist_com, label = 'debt', lw = 2)
    ax[1].plot(np.arange(N_simul), y_hist_com, label = 'Income',
               lw = 2, color = sb.color_palette()[3], alpha = .6, linestyle =
↪'--')
    ax[1].legend(loc = 'best', fontsize = 15)
    ax[1].axhline(0, color = 'k', lw = 1)
    ax[1].set_xlabel('Periods', fontsize = 13)

    plt.show()
```

Here's the resulting figure

Interpretation of graph

In the above graph, please note that:

- non financial income fluctuates in a stationary manner

- consumption is completely constant
- the consumer's debt fluctuates in a stationary manner; in fact, in this case because non-financial income is a first-order autoregressive process, the consumer's debt is an exact affine function (meaning linear plus a constant) of the consumer's non-financial 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 consumer's 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 Barro's *[Bar79]*

Government manipulation of Arrow securities prices

In *a linear quadratic Ramsey problem* 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

On-the-Job Search

Contents

- *On-the-Job Search*
  - *Overview*
  - *Model*
  - *Implementation*
  - *Solving for Policies*
  - *Exercises*
  - *Solutions*

Overview

In this section we solve a simple on-the-job search model

- based on *[LS12]*, 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

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

- ϕ_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

$$x_{t+1} = (1 - b_{t+1})G(x_t, \phi_t) + b_{t+1} \max\{G(x_t, \phi_t), U_{t+1}\} \tag{6.94}$$

Agent's objective: maximize expected discounted sum of wages via controls $\{s_t\}$ and $\{\phi_t\}$

Taking the expectation of $V(x_{t+1})$ and using (6.94), the Bellman equation for this problem can be written as

$$V(x) = \max_{s+\phi \leq 1} \left\{ x(1 - s - \phi) + \beta(1 - \pi(s))V[G(x, \phi)] + \beta\pi(s) \int V[G(x, \phi) \vee u]F(du) \right\}. \tag{6.95}$$

Here nonnegativity of s and ϕ 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 ϕ
2. search for a new job with better job-specific capital match via s

Since wages are $x(1 - s - \phi)$, marginal cost of investment via either ϕ or s is identical

Our risk neutral worker should focus on whatever instrument has the highest expected return

The relative expected return will depend on x

For example, suppose first that $x = 0.05$

- If $s = 1$ and $\phi = 0$, then since $G(x, \phi) = 0$, taking expectations of (6.94) gives expected next period capital equal to $\pi(s)\mathbb{E}U = \mathbb{E}U = 0.5$
- If $s = 0$ and $\phi = 1$, then next period capital is $G(x, \phi) = G(0.05, 1) \approx 0.23$

Both rates of return are good, but the return from search is better

Next suppose that $x = 0.4$

- If $s = 1$ and $\phi = 0$, then expected next period capital is again 0.5
- If $s = 0$ and $\phi = 1$, then $G(x, \phi) = G(0.4, 1) \approx 0.8$

Return from investment via ϕ dominates expected return from search

Combining these observations gives us two informal predictions:

1. At any given state x, the two controls ϕ and s will function primarily as substitutes — worker will focus on whichever instrument has the higher expected return
2. For sufficiently small x, search will be preferable to investment in job-specific human capital. For larger x, the reverse will be true

Now let's turn to implementation, and see if we can match our predictions

Implementation

The QuantEcon package provides some code for solving the DP problem described above

See in particular jv.py, which is repeated here for convenience

```
"""
Filename: jv.py

Authors: Thomas Sargent, John Stachurski

References
-----------

http://quant-econ.net/py/jv.html

"""
from textwrap import dedent
import sys
import numpy as np
from scipy.integrate import fixed_quad as integrate
from scipy.optimize import minimize
import scipy.stats as stats

# The SLSQP method is faster and more stable, but it didn't give the
# correct answer in python 3. So, if we are in python 2, use SLSQP, otherwise
# use the only other option (to handle constraints): COBYLA
if sys.version_info[0] == 2:
    method = "SLSQP"
else:
    # python 3
    method = "COBYLA"

epsilon = 1e-4  # A small number, used in the optimization routine


class JvWorker(object):
    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

    * x = human capital
    * s = search effort
    * :math:`\phi` = investment in human capital
    * :math:`\pi(s)` = probability of new offer given search level s
```

```
    * :math:`x(1 - \phi - s)` = wage
    * :math:`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)
        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, phi: A * (x * phi)**alpha)
        Transition function for human captial
    pi : function, optional(default=sqrt)
        Function mapping search effort (:math:`s \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

    """

    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, phi: A * (x * phi)**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 - epsilon))
        self.x_grid = np.linspace(epsilon, grid_max, grid_size)

    def __repr__(self):
        m = "JvWorker(A={a:g}, alpha={al:g}, beta={b:g}, grid_size={gs})"
        return m.format(a=self.A, al=self.alpha, b=self.beta,
                        gs=self.x_grid.size)

    def __str__(self):
        m = """\
        Jovanovic worker (on the job search):
```

```
        - A (parameter in human capital transition function)     : {a:g}
        - alpha (parameter in human capital transition function) : {al:g}
        - beta (parameter in human capital transition function)  : {b:g}
        - grid_size (number of grid points for human capital)    : {gs}
        - grid_max (maximum of grid for human capital)           : {gm:g}
    """
    return dedent(m.format(a=self.A, al=self.alpha, b=self.beta,
                           gs=self.x_grid.size, gm=self.x_grid.max()))

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 phi_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. === #
    G, pi, F, beta = self.G, self.pi, self.F, self.beta
    Vf = lambda x: np.interp(x, self.x_grid, V)
    N = len(self.x_grid)
    new_V, s_policy, phi_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 + phi <= 1
    c2 = lambda z: z[0] - epsilon      # used to enforce s >= epsilon
    c3 = lambda z: z[1] - epsilon      # used to enforce phi >= epsilon
    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, phi = z
                h = lambda u: Vf(np.maximum(G(x, phi), u)) * F.pdf(u)
                integral, err = integrate(h, a, b)
                q = pi(s) * integral + (1.0 - pi(s)) * Vf(G(x, phi))
                # == minus because we minimize == #
                return - x * (1.0 - phi - s) - beta * q

            # === either use SciPy solver === #
            if not brute_force:
                max_s, max_phi = minimize(w, guess, constraints=constraints,
                                          options={"disp": 0},
                                          method=method)["x"]
                max_val = -w((max_s, max_phi))

            # === or search on a grid === #
            else:
                search_grid = np.linspace(epsilon, 1.0, 15)
                max_val = -1.0
                for s in search_grid:
                    for phi in search_grid:
                        current_val = -w((s, phi)) if s + phi <= 1.0 else -1.0
                        if current_val > max_val:
                            max_val, max_s, max_phi = current_val, s, phi

            # === store results === #
            new_V[i] = max_val
            s_policy[i], phi_policy[i] = max_s, max_phi

        if return_policies:
            return s_policy, phi_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 $A(x\phi)^\alpha$

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

$$w(s, \phi) := -\left\{ x(1 - s - \phi) + \beta(1 - \pi(s))V[G(x, \phi)] + \beta\pi(s) \int V[G(x, \phi) \vee u]F(du) \right\} \tag{6.96}$$

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_grid`

But to evaluate the right-hand side of (6.96), we need a function, so we replace the arrays `V` and `x_grid` with a function `Vf` that gives linear interpolation of `V` on `x_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 ϕ

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 (6.96).

The function is minimized over all feasible (s, ϕ) 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

Solving for Policies

Let's plot the optimal policies and see what they look like

The code is in this file and looks as follows

```
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, phi_policy = wp.bellman_operator(V, return_policies=True)

# === plot policies === #
```

```
fig, ax = plt.subplots()
ax.set_xlim(0, max(wp.x_grid))
ax.set_ylim(-0.1, 1.1)
ax.plot(wp.x_grid, phi_policy, 'b-', label='phi')
ax.plot(wp.x_grid, s_policy, 'g-', label='s')
ax.set_xlabel("x")
ax.legend()
plt.show()
```

It produces the following figure

Fig. 6.1: 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

Exercises

Exercise 1

Let's look at the dynamics for the state process $\{x_t\}$ associated with these policies

The dynamics are given by (6.94) when ϕ_t and s_t are chosen according to the optimal policies, and $\mathbb{P}\{b_{t+1} = 1\} = \pi(s_t)$

Since the dynamics are random, analysis is a bit subtle

One way to do it is to plot, for each x in a relatively fine grid called `plot_grid`, a large number K of realizations of x_{t+1} given $x_t = x$. Plot this with one dot for each realization, in the form of a 45 degree diagram. Set

```
K = 50
plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = np.linspace(0, plot_grid_max, plot_grid_size)
fig, ax = plt.subplots()
ax.set_xlim(0, plot_grid_max)
ax.set_ylim(0, plot_grid_max)
```

By examining the plot, argue that under the optimal policies, the state x_t will converge to a constant value $\bar{x}$ close to unity

Argue that at the steady state, $s_t \approx 0$ and $\phi_t \approx 0.6$

Exercise 2

In the preceding exercise we found that s_t converges to zero and ϕ_t converges to about 0.6

Since these results were calculated at a value of β close to one, let's compare them to the best choice for an *infinitely* patient worker

Intuitively, an infinitely patient worker would like to maximize steady state wages, which are a function of steady state capital

You can take it as given—it's certainly true—that the infinitely patient worker does not search in the long run (i.e., $s_t = 0$ for large t)

Thus, given ϕ, steady state capital is the positive fixed point $x^*(\phi)$ of the map $x \mapsto G(x, \phi)$

Steady state wages can be written as $w^*(\phi) = x^*(\phi)(1 - \phi)$

Graph $w^*(\phi)$ with respect to ϕ, and examine the best choice of ϕ

Can you give a rough interpretation for the value that you see?

Solutions

```
import numpy as np
import matplotlib.pyplot as plt
import random
from quantecon import compute_fixed_point
from jv import JvWorker
```

Exercise 1

Here's code to produce the 45 degree diagram

```
wp = JvWorker(grid_size=25)
G, pi, F = wp.G, wp.pi, 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, phi_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)
phi = lambda y: np.interp(y, wp.x_grid, phi_policy)

def h(x, b, U):
    return (1 - b) * G(x, phi(x)) + b * max(G(x, phi(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)
ax.set_yticks(ticks)
ax.set_xlabel(r'$x_t$', fontsize=16)
ax.set_ylabel(r'$x_{t+1}$', 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) < pi(s(x)) else 0
        U = wp.F.rvs(1)
        y = h(x, b, U)
        ax.plot(x, y, 'go', alpha=0.25)

plt.show()
```

```
Computing value function
Computing policy functions
```

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(phi):
    return (wp.A * phi**wp.alpha)**(1 / (1 - wp.alpha))

phi_grid = np.linspace(0, 1, 100)
fig, ax = plt.subplots(figsize=(9, 7))
ax.set_xlabel(r'$\phi$', fontsize=16)
ax.plot(phi_grid, [xbar(phi) * (1 - phi) for phi in phi_grid], 'b-', label=r'
↪$w^*(\phi)$')
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 ϕ 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

CHAPTER

SEVEN

PART 7: TOPICS IN ECONOMIC DYNAMICS

Here we look at extensions to previous models and interesting dynamic models not previously discussed.

Modeling Career Choice

Contents

- *Modeling Career Choice*
  - *Overview*
  - *Model*
  - *Implementation:* `career.py`
  - *Exercises*
  - *Solutions*

Overview

Next we study a computational problem concerning career and job choices

The model is originally due to Derek Neal *[Nea99]*

This exposition draws on the presentation in *[LS12]*, section 6.5

Model features

- career and job within career both chosen to maximize expected discounted wage flow
- infinite horizon dynamic programming with two states variables

Model

In what follows we distinguish between a career and a job, where

- a *career* is understood to be a general field encompassing many possible jobs, and
- a *job* is understood to be a position with a particular firm

For workers, wages can be decomposed into the contribution of job and career

- $w_t = \theta_t + \epsilon_t$, where
  - θ_t is contribution of career at time t
  - ϵ_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 (θ_t, ϵ_t) — referred to hereafter as "stay put"
- retain a current career θ_t but redraw a job ϵ_t — referred to hereafter as "new job"
- redraw both a career θ_t and a job ϵ_t — referred to hereafter as "new life"

Draws of θ and ϵ are independent of each other and past values, with

- $\theta_t \sim F$
- $\epsilon_t \sim G$

Notice that the worker does not have the option to retain a job but redraw a career — starting a new career always requires starting a new job

A young worker aims to maximize the expected sum of discounted wages

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t w_t \tag{7.1}$$

subject to the choice restrictions specified above

Let $V(\theta, \epsilon)$ denote the value function, which is the maximum of (7.1) over all feasible (career, job) policies, given the initial state (θ, ϵ)

The value function obeys

$$V(\theta, \epsilon) = \max\{I, II, III\},$$

where

$$\begin{aligned}
I &= \theta + \epsilon + \beta V(\theta, \epsilon) \\
II &= \theta + \int \epsilon' G(d\epsilon') + \beta \int V(\theta, \epsilon') G(d\epsilon') \\
III &= \int \theta' F(d\theta') + \int \epsilon' G(d\epsilon') + \beta \int \int V(\theta', \epsilon') G(d\epsilon') F(d\theta')
\end{aligned}$$

Evidently I, II and III correspond to "stay put", "new job" and "new life", respectively

Parameterization

As in *[LS12]*, section 6.5, we will focus on a discrete version of the model, parameterized as follows:

- both θ and ϵ take values in the set `np.linspace(0, B, 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 `np.linspace(0, B, N)`

A very useful family of discrete distributions is the Beta-binomial family, with probability mass function

$$p(k \,|\, n, a, b) = \binom{n}{k} \frac{B(k+a, n-k+b)}{B(a,b)}, \qquad k = 0, \ldots, n$$

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

Here's a figure showing the effect of different shape parameters when $n = 50$

The code that generated this figure can be found here

Implementation: `career.py`

The code for solving the DP problem described above is found in this file, which is repeated here for convenience

```
"""
Filename: career.py

Authors: Thomas Sargent, John Stachurski

A class to solve the career / job choice model due to Derek Neal.

References
----------

http://quant-econ.net/py/career.html

..  [Neal1999] Neal, D. (1999). The Complexity of Job Mobility among
    Young Men, Journal of Labor Economics, 17(2), 237-261.

"""
import numpy as np
from quantecon.distributions import BetaBinomial
```

```
class CareerWorkerProblem(object):
    """
    An instance of the class is an object with data on a particular
    problem of this type, including probabilites, 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 epsilon and theta
    N : scalar(int), optional(default=50)
        Number of possible realizations for both epsilon 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
    epsilon : array_like(float, ndim=1)
        A grid of values from 0 to B
    F_probs : array_like(float, ndim=1)
        The probabilities of different values for F
    G_probs : array_like(float, ndim=1)
        The probabilities of different values for G
    F_mean : scalar(float)
        The mean of the distribution for F
    G_mean : scalar(float)
        The mean of the distribution for G

    """

    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.epsilon = np.linspace(0, B, N)   # set of epsilon values
        self.F_probs = BetaBinomial(N-1, F_a, F_b).pdf()
        self.G_probs = BetaBinomial(N-1, G_a, G_b).pdf()
        self.F_mean = np.sum(self.theta * self.F_probs)
        self.G_mean = np.sum(self.epsilon * self.G_probs)

        # Store these parameters for str and repr methods
```

```
        self._F_a, self._F_b = F_a, F_b
        self._G_a, self._G_b = G_a, G_b


    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: :math:`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.epsilon[j] + self.beta * v[i, j]

                # new job
                v2 = (self.theta[i] + self.G_mean + self.beta *
                      np.dot(v[i, :], self.G_probs))

                # new life
                v3 = (self.G_mean + self.F_mean + self.beta *
                      np.dot(self.F_probs, np.dot(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: :math:`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 :math:`(\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)
        for i in range(self.N):
            for j in range(self.N):
                v1 = self.theta[i] + self.epsilon[j] + self.beta * v[i, j]
                v2 = (self.theta[i] + self.G_mean + self.beta *
                      np.dot(v[i, :], self.G_probs))
                v3 = (self.G_mean + self.F_mean + self.beta *
                      np.dot(self.F_probs, np.dot(v, self.G_probs)))
                if v1 > max(v2, v3):
                    action = 1
                elif v2 > max(v1, v3):
                    action = 2
                else:
                    action = 3
                policy[i, j] = action

        return policy
```

The code defines

- a class `CareerWorkerProblem` that
  - encapsulates all the details of a particular parameterization
  - implement 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 (7.2), 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 *[LS12]*, 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

Fig. 7.1: Value function with uniform probabilities

The code used to produce this plot was career/career_vf_plot.py

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

Exercises

Exercise 1

Using the default parameterization in the class `CareerWorkerProblem`, generate and plot typical sample paths for θ and ϵ when the worker follows the optimal policy

In particular, modulo randomness, reproduce the following figure (where the horizontal axis represents time)

Hint: To generate the draws from the distributions F and G, use 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

$$T^* := \text{the first point in time from which the worker's job no longer changes}$$

Evidently, the worker's job becomes permanent if and only if (θ_t, ϵ_t) enters the "stay put" region of (θ, ϵ) space

Letting S denote this region, T^* can be expressed as the first passage time to S under the optimal policy:

$$T^* := \inf\{t \geq 0 \,|\, (\theta_t, \epsilon_t) \in S\}$$

Collect 25,000 draws of this random variable and compute the median (which should be about 7)

Repeat the exercise with $\beta = 0.99$ and interpret the change

Exercise 3

As best you can, reproduce *the figure showing the optimal policy*

Hint: The `get_greedy()` function 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.

Solutions

```
import matplotlib.pyplot as plt
from quantecon import DiscreteRV, compute_fixed_point
```

Exercise 1

Simulate job / career paths

In reading the code, recall that `optimal_policy[i, j]` = policy at (θ_i, ϵ_j) = either 1, 2 or 3; meaning 'stay put', 'new job' and 'new life'

```
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 = DiscreteRV(wp.F_probs)
G = DiscreteRV(wp.G_probs)

def gen_path(T=20):
    i = j = 0
    theta_index = []
    epsilon_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(G.draw())
        else:                            # New life
            i, j  = int(F.draw()), int(G.draw())
        theta_index.append(i)
        epsilon_index.append(j)
    return wp.theta[theta_index], wp.epsilon[epsilon_index]

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
for ax in axes:
    theta_path, epsilon_path = gen_path()
    ax.plot(epsilon_path, label='epsilon')
    ax.plot(theta_path, label='theta')
    ax.legend(loc='lower right')

plt.show()
```

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 = DiscreteRV(wp.F_probs)
G = DiscreteRV(wp.G_probs)

def gen_first_passage_time():
    t = 0
    i = j = 0
    while 1:
        if optimal_policy[i, j] == 1:    # Stay put
            return t
        elif optimal_policy[i, j] == 2:  # New job
            j = int(G.draw())
```

```
        else:                               # New life
            i, j  = int(F.draw()), int(G.draw())
        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))
```

```
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
25           1.460e+00      2.550e-01
50           4.050e-01      5.062e-01
75           1.123e-01      7.576e-01
100          3.116e-02      1.010e+00
125          8.644e-03      1.263e+00
150          2.398e-03      1.515e+00
168          9.524e-04      1.695e+00
Converged in 168 steps
7.0
```

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, 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

```
from matplotlib import cm

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.theta, wp.epsilon)
lvls=(0.5, 1.5, 2.5, 3.5)
ax.contourf(tg, eg, optimal_policy.T, levels=lvls, cmap=cm.winter, alpha=0.5)
ax.contour(tg, eg, optimal_policy.T, colors='k', levels=lvls, linewidths=2)
ax.set_xlabel('theta', fontsize=14)
ax.set_ylabel('epsilon', 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()
```

```
Iteration    Distance       Elapsed (seconds)
--------------------------------------------
25           1.460e+00      2.539e-01
50           4.050e-01      5.062e-01
75           1.123e-01      7.587e-01
100          3.116e-02      1.010e+00
125          8.644e-03      1.263e+00
150          2.398e-03      1.519e+00
168          9.524e-04      1.699e+00
Converged in 168 steps
```

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 ϵ has become more concentrated around the mean, making high-paying jobs less realistic

Dynamic Stackelberg Problems

Contents

- *Dynamic Stackelberg Problems*
  - *Overview*
  - *The Stackelberg Problem*
  - *Solving the Stackelberg Problem*
  - *Shadow prices*
  - *A Large Firm With a Competitive Fringe*
  - *Concluding Remarks*
  - *Exercises*

Overview

Previous lectures including the *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

- The time $t \geq 0$ actions of decision makers called followers depend on the 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 is sometimes 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

- Variables that encode *history dependence* appear in optimal decision rules of the Stackelberg leader at times $t \geq 1$
- The Stackelberg leader's decision rules for $t = 0$ and $t \geq 1$ have distinct state variables
- These properties of the Stackelberg leader's decision rules are symptoms of the *time inconsistency of optimal government plans*
- An expression of time inconsistency is that optimal decision rules at times $t \geq 1$ depend not only on the natural state variables at time t

Examples of time inconsistent optimal rules are those of a large agent (e.g., a government) who

- confronts a competitive market composed of many small private agents, and in which
- the private agents' decisions at each date are influenced by their *forecasts* of the government's future actions

The *rational expectations* equilibrium concept plays an essential role

A rational expectations restriction requires that when choosing its future actions, the Stackelberg leader also chooses the followers' expectations about those actions

The Stackelberg leader understands and exploits that

In a rational expectations equilibrium, the Stackelberg leader must confirm private agents' earlier forecasts of the Stackelberg leader's time t 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

An important lesson to be taught in this lecture is that if the natural state variables are properly augmented with additional forward-looking state variables inherited from the past, then this class of problems can still be made recursive

This lesson yields substantial insights and affords significant computational advantages

This lecture displays these principles within the tractable framework of linear quadratic problems

It is based on chapter 19 of *[LS12]*

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'll give an application with another interpretation of these two decision makers

Let z_t be an $n_z \times 1$ vector of natural state variables, x_t an $n_x \times 1$ vector of endogenous forward-looking variables that are physically free to jump at t, and u_t a vector of government instruments

The z_t vector is inherited physically from the past

But x_t is inherited from the past not physically but as a consequence of earlier promises

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[1]

$$r(y, u) = y'Ry + u'Qu \tag{7.2}$$

Subject to an initial condition for z_0, but not for x_0, a government wants to maximize

$$-\sum_{t=0}^{\infty} \beta^t r(y_t, u_t) \tag{7.3}$$

The government makes policy in light of the model

$$\begin{bmatrix} I & 0 \\ G_{21} & G_{22} \end{bmatrix} \begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \hat{A}_{11} & \hat{A}_{12} \\ \hat{A}_{21} & \hat{A}_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + \hat{B}u_t \tag{7.4}$$

We assume that the matrix on the left is invertible, so that we can multiply both sides of (7.4) by its inverse to obtain

$$\begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + Bu_t \tag{7.5}$$

or

$$y_{t+1} = Ay_t + Bu_t \tag{7.6}$$

The private sector's behavior is summarized by the second block of equations of (7.5) or (7.6)

These 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 sector's best response to the sequence of actions by the government.

The government maximizes (7.3) by choosing sequences $\{u_t, x_t, z_{t+1}\}_{t=0}^{\infty}$ subject to (7.6) and an initial condition for z_0

Note that we have an intitial condition for z_0 but not for x_0

[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).

x_0 is among the variables to be chosen at time 0

The government uses its understanding of the responses restricted by (7.6) to manipulate private sector actions

To indicate the features of the problem that make x_t a vector of forward-looking variables, write the second block of system (7.4) as

$$x_t = \phi_1 z_t + \phi_2 z_{t+1} + \phi_3 u_t + \phi_0 x_{t+1}, \tag{7.7}$$

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 ϕ_0 are bounded in modulus by $\beta^{-.5}$.

This stability condition makes equation (7.7) explosive if solved 'backwards' but stable if solved 'forwards'.

See the appendix of chapter 2 of *[LS12]*

So we solve equation (7.7) forward to get

$$x_t = \sum_{j=0}^{\infty} \phi_0^j \left[\phi_1 z_{t+j} + \phi_2 z_{t+j+1} + \phi_3 u_{t+j} \right]. \tag{7.8}$$

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 (7.8).

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 (7.5) or (7.6) 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

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 (7.6), where ϵ_{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 (7.3) 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 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 (7.5) as exhibited by (7.8)

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 the following two subproblems:

Subproblem 1

$$v(y_0) = \max_{(\vec{y}_1, \vec{u}_0) \in \Omega(y_0)} -\sum_{t=0}^{\infty} \beta^t r(y_t, u_t) \tag{7.9}$$

Subproblem 2

$$w(z_0) = \max_{x_0} v(y_0) \tag{7.10}$$

Subproblem 1 is solved first, once-and-for-all at time 0, tentatively taking the vector of forward-looking variables x_0 as given

Then subproblem 2 is solved for 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

[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*

Two Bellman equations

We now describe Bellman equations for $v(y)$ and $w(z_0)$

Subproblem 1

The value function $v(y)$ in subproblem 1 satisfies the Bellman equation

$$v(y) = \max_{u,y^*} \{-r(y,u) + \beta v(y^*)\} \tag{7.11}$$

where the maximization is subject to

$$y^* = Ay + Bu \tag{7.12}$$

and y^* denotes next period's value.

Substituting $v(y) = -y'Py$ into Bellman equation (7.11) gives

$$-y'Py = \max_{u,y^*} \left\{-y'Ry - u'Qu - \beta y^{*\prime}Py^*\right\}$$

which as in lecture *linear regulator* gives rise to the algebraic matrix Riccati equation

$$P = R + \beta A'PA - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA \tag{7.13}$$

and the optimal decision rule coefficient vector

$$F = \beta(Q + \beta B'PB)^{-1}B'PA, \tag{7.14}$$

where the optimal decision rule is

$$u_t = -Fy_t. \tag{7.15}$$

Subproblem 2

The value function $v(y_0)$ satisfies

$$v(y_0) = -z_0'P_{11}z_0 - 2x_0'P_{21}z_0 - x_0'P_{22}x_0 \tag{7.16}$$

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

$$x_0 = -P_{22}^{-1}P_{21}z_0. \tag{7.17}$$

Summary

We solve the Stackelberg problem by

- formulating a particular optimal linear regulator
- solving the associated matrix Riccati equation (7.13) for P
- computing F
- then partitioning P to obtain representation (7.17)

Manifestation of time inconsistency

We have seen that for $t \geq 0$ the optimal decision rule for the Stackelberg leader has the form

$$u_t = -Fy_t$$

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 (7.17) implies that

$$u_0 = (f_{11} - f_{12}P_{22}^{-1}P_{2,1})z_0 \tag{7.18}$$

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 only 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 leader's 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

Shadow prices

The history dependence of the government's plan can be expressed in the dynamics of Lagrange multipliers μ_x on the last n_x equations of (7.4) or (7.5)

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 μ_x take nonzero values thereafter, reflecting future costs to the government of confirming the private sector's earlier expectations about its time t actions

From the *linear regulator* lecture, the formula $\mu_t = Py_t$ for the vector of shadow prices on the transition equations is

$$\mu_t = \begin{bmatrix} \mu_{zt} \\ \mu_{xt} \end{bmatrix}$$

The shadow price μ_{xt} on the forward-looking variables x_t evidently equals

$$\mu_{xt} = P_{21} z_t + P_{22} x_t. \tag{7.19}$$

So (7.17) is equivalent with

$$\mu_{x0} = 0. \tag{7.20}$$

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[3]

The costs of production are $\mathcal{C}_t = eQ_t + .5gQ_t^2 + .5c(Q_{t+1} - Q_t)^2$ for the large firm and $\sigma_t = dq_t + .5hq_t^2 + .5c(q_{t+1} - q_t)^2$ 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

$$p_t = A_0 - A_1(Q_t + \overline{q}_t) + v_t, \tag{7.21}$$

where A_0, A_1 are both positive and v_t is a disturbance to demand governed by

$$v_{t+1} = \rho v_t + C_\epsilon \check{\epsilon}_{t+1} \tag{7.22}$$

[3] *[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.

where $|\rho| < 1$ and $\check{\epsilon}_{t+1}$ is an IID sequence of random variables with mean zero and variance 1

In (7.21), $\overline{q}_t$ is equilibrium output of the representative competitive firm

In equilibrium, $\overline{q}_t = q_t$, but we must distinguish between q_t and $\overline{q}_t$ in posing the optimum problem of a competitive firm

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

$$E_0 \sum_{t=0}^{\infty} \beta^t \left\{ p_t q_t - \sigma_t \right\}, \quad \beta \in (0,1) \tag{7.23}$$

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 firm's control at t

The first-order conditions for maximizing (7.23) are

$$i_t = E_t \beta i_{t+1} - c^{-1} \beta h q_{t+1} + c^{-1} \beta E_t (p_{t+1} - d) \tag{7.24}$$

for $t \geq 0$

We appeal to a *certainty equivalence principle* to justify working with a non-stochastic version of (7.24) formed by dropping the expectation operator and the random term $\check{\epsilon}_{t+1}$ from (7.22)

We use a method of *[Sar79]* and *[Tow83]*[4]

We shift (7.21) forward one period, replace conditional expectations with realized values, use (7.21) to substitute for p_{t+1} in (7.24), and set $q_t = \overline{q}_t$ and $i_t = \bar{i}_t$ for all $t \geq 0$ to get

$$\bar{i}_t = \beta \bar{i}_{t+1} - c^{-1} \beta h \overline{q}_{t+1} + c^{-1} \beta (A_0 - d) - c^{-1} \beta A_1 \overline{q}_{t+1} - c^{-1} \beta A_1 Q_{t+1} + c^{-1} \beta v_{t+1} \tag{7.25}$$

Given sufficiently stable sequences $\{Q_t, v_t\}$, we could solve (7.25) and $\bar{i}_t = \overline{q}_{t+1} - \overline{q}_t$ to express the competitive fringe's output sequence as a function of the (tail of the) monopolist's output sequence

(This would be a version of representation (7.8))

It is this feature that makes the monopolist's 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

[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 firm's first-order conditions to get a difference equation in capital.

The monopolist's problem

The monopolist views the sequence of the competitive firm's Euler equations as constraints on its own opportunities

They are *implementability constraints* on the monopolist's choices

Including the implementability constraints, we can represent the constraints in terms of the transition law facing the monopolist:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ A_0 - d & 1 & -A_1 & -A_1 - h & c \end{bmatrix} \begin{bmatrix} 1 \\ v_{t+1} \\ Q_{t+1} \\ \overline{q}_{t+1} \\ \bar{i}_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \rho & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & \frac{c}{\beta} \end{bmatrix} \begin{bmatrix} 1 \\ v_t \\ Q_t \\ \overline{q}_t \\ \bar{i}_t \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} u_t \tag{7.26}$$

where $u_t = Q_{t+1} - Q_t$ is the control of the monopolist at time t

The last row portrays the implementability constraints (7.25)

Represent (7.26) as

$$y_{t+1} = Ay_t + Bu_t \tag{7.27}$$

Although we have included the competitive fringe's choice variable $\bar{i}_t$ as a component of the "state" y_t in the monopolist's transition law (7.27), $\bar{i}_t$ is actually a "jump" variable

Nevertheless, the analysis above implies that the solution of the large firm's problem is encoded in the Riccati equation associated with (7.27) as the transition law

Let's decode it

To match our general setup, we partition y_t as $y_t' = \begin{bmatrix} z_t' & x_t' \end{bmatrix}$ where $z_t' = \begin{bmatrix} 1 & v_t & Q_t & \overline{q}_t \end{bmatrix}$ and $x_t = \bar{i}_t$

The monopolist's problem is

$$\max_{\{u_t, p_{t+1}, Q_{t+1}, \overline{q}_{t+1}, \bar{i}_t\}} \sum_{t=0}^{\infty} \beta^t \left\{ p_t Q_t - \mathcal{C}_t \right\}$$

subject to the given initial condition for z_0, equations (7.21) and (7.25) and $\bar{i}_t = \overline{q}_{t+1} - \overline{q}_t$, 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 (7.21) and the implementability constraint (7.25) that describes the best responses of firms in the competitive fringe

By substituting (7.21) into the above objective function, the monopolist's problem can be expressed as

$$\max_{\{u_t\}} \sum_{t=0}^{\infty} \beta^t \left\{ (A_0 - A_1(\overline{q}_t + Q_t) + v_t) Q_t - eQ_t - .5gQ_t^2 - .5cu_t^2 \right\} \tag{7.28}$$

subject to (7.27)

This can be written

$$\max_{\{u_t\}} -\sum_{t=0}^{\infty} \beta^t \left\{ y_t' R y_t + u_t' Q u_t \right\} \tag{7.29}$$

subject to (7.27) where

$$R = -\begin{bmatrix} 0 & 0 & \frac{A_0 - e}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 \\ \frac{A_0 - e}{2} & \frac{1}{2} & -A_1 - .5g & -\frac{A_1}{2} & 0 \\ 0 & 0 & -\frac{A_1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

and $Q = \frac{c}{2}$

Under the Stackelberg plan, $u_t = -Fy_t$, which implies that the evolution of y under the Stackelberg plan as

$$\overline{y}_{t+1} = (A - BF)\overline{y}_t \tag{7.30}$$

where $\overline{y}_t = \begin{bmatrix} 1 & v_t & Q_t & \overline{q}_t & \overline{i}_t \end{bmatrix}'$

Recursive formulation of a follower's problem

We now make use of a "Big K, little k" trick (see *rational expectations equilibrium*) to formulate a recursive version of a follower's problem cast in terms of an ordinary Bellman equation

The individual firm faces $\{p_t\}$ as a price taker and believes

$$\begin{aligned} p_t &= a_0 - a_1 Q_t - a_1 \overline{q}_t + v_t \\ &\equiv E_p \left[\overline{y}_t \right] \end{aligned} \tag{7.31}$$

(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

$$\begin{bmatrix} \overline{y}_{t+1} \\ q_{t+1} \end{bmatrix} = \begin{bmatrix} A - BF & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \overline{y}_t \\ q_t \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} i_t \tag{7.32}$$

We calculated F and therefore $A - BF$ earlier

We can restate the optimization problem of the representative competitive firm

The firm takes $\overline{y}_t$ as an exogenous process and chooses an output plan $\{q_t\}$ to maximize

$$E_0 \sum_{t=0}^{\infty} \beta^t \left\{ p_t q_t - \sigma_t \right\}, \quad \beta \in (0,1)$$

subject to q_0 given the law of motion (7.30) and the price function (7.31) and where the costs are still $\sigma_t = dq_t + .5hq_t^2 + .5c(q_{t+1} - q_t)^2$

The representative firm's 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 firm's decision rule can be represented as

$$i_t = -F_s \begin{bmatrix} 1 \\ v_t \\ Q_t \\ \overline{q}_t \\ \overline{i}_t \\ q_t \end{bmatrix} \tag{7.33}$$

Now let's stare at the decision rule (7.33) 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 follower's choice problem

- We want decision rule (7.33) to have the property that $i_t = \overline{i}_t$ when we evaluate it at $q_t = \overline{q}_t$

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 firm's optimization problem

- We want $q_t = \overline{q}_t$
- We want $i_t = \overline{i}_t$

Numerical example

We computed the optimal Stackelberg plan for parameter settings $A_0, A_1, \rho, C_\epsilon, c, d, e, g, h, \beta = 100, 1, .8, .2, 1, 20, 20, .2, .2, .95$[5]

For these parameter values the monopolist's decision rule is

$$u_t = (Q_{t+1} - Q_t) = \begin{bmatrix} 83.98 & 0.78 & -0.95 & -1.31 & -2.07 \end{bmatrix} \begin{bmatrix} 1 \\ v_t \\ Q_t \\ \overline{q}_t \\ \overline{i}_t \end{bmatrix}$$

for $t \geq 0$

[5] These calculations were performed by the Python program from QuantEcon.applications in dyn_stack/oligopoly.py.

and

$$x_0 \equiv \bar{i}_0 = \begin{bmatrix} 31.08 & 0.29 & -0.15 & -0.56 \end{bmatrix} \begin{bmatrix} 1 \\ v_0 \\ Q_0 \\ \bar{q}_0 \end{bmatrix}$$

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 monopolist's optimal plan

The recursive representation of the decision rule for a representative fringe firm is

$$i_t = \begin{bmatrix} 0 & 0 & 0 & .34 & 1 & -.34 \end{bmatrix} \begin{bmatrix} 1 \\ v_t \\ Q_t \\ \bar{q}_t \\ \bar{i}_t \\ q_t \end{bmatrix},$$

which we have computed by solving the appropriate linear-quadratic dynamic programming problem described above

Notice that, as expected, $i_t = \bar{i}_t$ when we evaluate this decision rule at $q_t = \bar{q}_t$

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 another example in the lectures on *history dependent policies*, *optimal taxation with state-contingent debt*, and *optimal taxtion without state-contingent debt*

There are many more such problems - see chapters 20-24 of *[LS12]*

Exercises

Exercise 1

There is no uncertainty

[6] For another application of the techniques in this lecture and how they related to the method recommended by *[KP80b]*, please see *this lecture* .

For $t \geq 0$, a monetary authority sets the growth of (the log of) money according to

$$m_{t+1} = m_t + u_t \tag{7.34}$$

subject to the initial condition $m_0 > 0$ given

The demand for money is

$$m_t - p_t = -\alpha(p_{t+1} - p_t) \tag{7.35}$$

where $\alpha > 0$ and p_t is the log of the price level

Equation (7.34) can be interpreted as the Euler equation of the holders of money

a. Briefly interpret how (7.34) makes the demand for real balances vary inversely with the expected rate of inflation. Temporarily (only for this part of the exercise) drop (7.34) and assume instead that $\{m_t\}$ is a given sequence satisfying $\sum_{t=0}^{\infty} m_t^2 < +\infty$. Solve the difference equation (7.34) "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

$$-\sum_{t=0}^{\infty} .95^t \left[(p_t - \overline{p})^2 + u_t^2 + .00001 m_t^2\right] \tag{7.36}$$

Assume that $m_0 = 10, \alpha = 5, \bar{p} = 1$

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 $.00001 m_t^2$ 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

$$\sum_{t=0}^{\infty} \beta^t \left\{-.5(b - c_t)^2\right\} \tag{7.37}$$

where $\beta \in (0, 1)$, $b = 30$, and c_t is time t consumption

The consumer faces a sequence of budget constraints

$$c_t + a_{t+1} = (1+r)a_t + y_t - \tau_t \tag{7.38}$$

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

$$y_t = \rho y_{t-1}, \quad t \geq 1, \tag{7.39}$$

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

$$\sum_{t=0}^{\infty} \beta^t \left\{ -.5(c_t - b)^2 - \tau_t^2 \right\} \tag{7.40}$$

over $\{c_t, \tau_t\}_{t=0}^{\infty}$ subject to the implementability constraints in (7.38) for $t \geq 0$ and

$$\lambda_t = \beta(1+r)\lambda_{t+1} \tag{7.41}$$

for $t \geq 0$, where $\lambda_t \equiv (b - c_t)$

a. Argue that (7.41) is the Euler equation for a consumer who maximizes (7.37) subject to (7.38), taking $\{\tau_t\}$ as a given sequence

b. Formulate the planner's problem as a Stackelberg problem

c. For $\beta = .95, b = 30, \beta(1+r) = .95$, 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 τ_t

History Dependent Public Policies

Contents

- *History Dependent Public Policies*
  - *Overview*
  - *Two Sources of History Dependence*

- *Competitive equilibrium*
- *Ramsey Problem*
- *Two Subproblems*
- *Time Inconsistency*
- *Credible Policy*
- *Concluding remarks*

Overview

This lecture describes history-dependent public policies and some of their representations

History dependent policies are decision rules that depend on the entire past history of the state variables

History dependent policies naturally emerge in *Ramsey problems*

A Ramsey planner (typically interpreted as a government) devises a plan of actions at time $t = 0$ to follow at all future dates and for all contingencies

In order to make a plan, he takes as given Euler equations expressing private agents' first-order necessary conditions

He also takes into account that his *future* actions affect earlier decisions by private agents, an avenue opened up by the maintained assumption of *rational expectations*

Another setting in which history dependent policies naturally emerge is where instead of a Ramsey planner there is a *sequence* of government administrators whose time t member takes as given the policies used by its successors

We study these ideas in the context of a model in which a benevolent tax authority is forced

- to raise a prescribed present value of revenues
- to do so by imposing a distorting flat rate tax on the output of a competitive representative firm

The firm faces costs of adjustment and lives within a competitive equilibrium, which in turn imposes restrictions on the tax authority[1]

References

The presentation below is based on a recent paper by Evans and Sargent *[ES13]*

Regarding techniques, we will make use of the methods described in

1. the *linear regulator lecture*
2. the *solving LQ dynamic Stackelberg problems lecture*

[1] We could also call a competitive equilibrium a rational expectations equilibrium.

Two Sources of History Dependence

We compare two timing protocols

1. An infinitely lived benevolent tax authority solves a Ramsey problem
2. There is a sequence of tax authorities, each choosing only a time t tax rate

Under both timing protocols, optimal tax policies are *history-dependent*

But history dependence captures different economic forces across the two timing protocols

In the first timing protocol, history dependence expresses the *time-inconsistency of the Ramsey plan*

In the second timing protocol, history dependence reflects the unfolding of constraints that assure that a time t government administrator wants to confirm the representative firm's expectations about government actions

We describe recursive representations of history-dependent tax policies under both timing protocols

Ramsey Timing Protocol

The first timing protocol models a policy maker who can be said to 'commit', choosing a sequence of tax rates once-and-for-all at time 0

Sequence of Governments Timing Protocol

For the second timing protocol we use the notion of a *sustainable plan* proposed in *[CK90]*, also referred to as a *credible public policy* in *[Sto89]*

A key idea here is that history-dependent policies can be arranged so that, when regarded as a representative firm's forecasting functions, they confront policy makers with incentives to confirm them

We follow Chang *[Cha98]* in expressing such history-dependent plans recursively

Credibility considerations contribute an additional auxiliary state variable in the form of a promised value to the planner

It expresses how decisions must unfold to give the government the incentive to confirm private sector expectations when the government chooses sequentially

---

Note: We occasionally hear confusion about the consequences of recursive representations of government policies under our two timing protocols. It is incorrect to regard a recursive representation of the Ramsey plan as in any way 'solving a time-inconsistency problem'. On the contrary, the evolution of the auxiliary state variable that augments the authentic ones under our first timing protocol ought to be viewed as *expressing* the time-inconsistency of a Ramsey plan. Despite that, in literatures about practical monetary policy one sometimes hears interpretations that sell Ramsey plans in settings where our sequential timing protocol is the one that more accurately characterizes decision making. Please beware of discussions that toss around claims about credibility if you don't also see recursive representations of policies with the complete list of state variables appearing in our *[Cha98]* -like analysis that we present *below*.

Competitive equilibrium

A representative competitive firm sells output q_t at price p_t when market-wide output is Q_t

The market as a whole faces a downward sloping inverse demand function

$$p_t = A_0 - A_1 Q_t, \quad A_0 > 0, A_1 > 0 \tag{7.42}$$

The representative firm

- has given initial condition q_0
- endures quadratic adjustment costs $\frac{d}{2}(q_{t+1} - q_t)^2$
- pays a flat rate tax τ_t per unit of output
- treats $\{p_t, \tau_t\}_{t=0}^{\infty}$ as exogenous
- chooses $\{q_{t+1}\}_{t=0}^{\infty}$ to maximize

$$\sum_{t=0}^{\infty} \beta^t \{p_t q_t - \frac{d}{2}(q_{t+1} - q_t)^2 - \tau_t q_t\} \tag{7.43}$$

Let $u_t := q_{t+1} - q_t$ be the firm's 'control variable' at time t

First-order conditions for the representative firm's problem are

$$u_t = \frac{\beta}{d} p_{t+1} + \beta u_{t+1} - \frac{\beta}{d}\tau_{t+1}, \quad t = 0, 1, \ldots \tag{7.44}$$

To compute a competitive equilibrium, it is appropriate to take (7.44), eliminate p_t in favor of Q_t by using (7.42), and then set $q_t = Q_t$

This last step *makes the representative firm be representative*[2]

We arrive at

$$u_t = \frac{\beta}{d}(A_0 - A_1 Q_{t+1}) + \beta u_{t+1} - \frac{\beta}{d}\tau_{t+1} \tag{7.45}$$

$$Q_{t+1} = Q_t + u_t \tag{7.46}$$

Notation: For any scalar x_t, let $\vec{x} = \{x_t\}_{t=0}^{\infty}$

Given a tax sequence $\{\tau_{t+1}\}_{t=0}^{\infty}$, a **competitive equilibrium** is a price sequence $\vec{p}$ and an output sequence $\vec{Q}$ that satisfy (7.42), (7.45), and (7.46)

For any sequence $\vec{x} = \{x_t\}_{t=0}^{\infty}$, the sequence $\vec{x}_1 := \{x_t\}_{t=1}^{\infty}$ is called the **continuation sequence** or simply the **continuation**

Note that a competitive equilibrium consists of a first period value $u_0 = Q_1 - Q_0$ and a continuation competitive equilibrium with initial condition Q_1

Also, a continuation of a competitive equilibrium is a competitive equilibrium

Following the lead of *[Cha98]*, we shall make extensive use of the following property:

[2] It is important not to set $q_t = Q_t$ prematurely. To make the firm a price taker, this equality should be imposed *after* and not *before* solving the firm's optimization problem.

- A continuation $\vec{\tau}_1 = \{\tau_t\}_{t=1}^{\infty}$ of a tax policy $\vec{\tau}$ influences u_0 via (7.45) entirely through its impact on u_1

A continuation competitive equilibrium can be indexed by a u_1 that satisfies (7.45)

In the spirit of *[KP80a]* , we shall use u_{t+1} to describe what we shall call a **promised marginal value** that a competitive equilibrium offers to a representative firm[3]

Define $Q^t := [Q_0, \ldots, Q_t]$

A **history-dependent tax policy** is a sequence of functions $\{\sigma_t\}_{t=0}^{\infty}$ with σ_t mapping Q^t into a choice of τ_{t+1}

Below, we shall

- Study history-dependent tax policies that either solve a Ramsey plan or are credible
- Describe recursive representations of both types of history-dependent policies

Ramsey Problem

The planner's objective is cast in terms of consumer surplus net of the firm's adjustment costs

Consumer surplus is

$$\int_0^Q (A_0 - A_1 x) dx = A_0 Q - \frac{A_1}{2} Q^2$$

Hence the planner's one-period return function is

$$A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 \tag{7.47}$$

At time $t = 0$, a Ramsey planner faces the intertemporal budget constraint

$$\sum_{t=1}^{\infty} \beta^t \tau_t Q_t = G_0 \tag{7.48}$$

Note that (7.48) forbids taxation of initial output Q_0

The **Ramsey problem** is to choose a tax sequence $\vec{\tau}_1$ and a competitive equilibrium outcome $(\vec{Q}, \vec{u})$ that maximize

$$\sum_{t=0}^{\infty} \beta^t \left\{ A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 \right\} \tag{7.49}$$

subject to (7.48)

Thus, the Ramsey timing protocol is:

[3] We could instead, perhaps with more accuracy, define a promised marginal value as $\beta(A_0 - A_1 Q_{t+1}) - \beta\tau_{t+1} + u_{t+1}/\beta$, since this is the object to which the firm's first-order condition instructs it to equate to the marginal cost du_t of $u_t = q_{t+1} - q_t$. This choice would align better with how Chang *[Cha98]* chose to express his competitive equilibrium recursively. But given (u_t, Q_t), the representative firm knows (Q_{t+1}, τ_{t+1}), so it is adequate to take u_{t+1} as the intermediate variable that summarizes how $\vec{\tau}_{t+1}$ affects the firm's choice of u_t.

1. At time 0, knowing (Q_0, G_0), the Ramsey planner chooses $\{\tau_{t+1}\}_{t=0}^{\infty}$
2. Given $\left(Q_0, \{\tau_{t+1}\}_{t=0}^{\infty}\right)$, a competitive equilibrium outcome $\{u_t, Q_{t+1}\}_{t=0}^{\infty}$ emerges

---

Note: In bringing out the timing protocol associated with a Ramsey plan, we run head on into a set of issues analyzed by Bassetto *[Bas05]*. This is because our definition of the Ramsey timing protocol doesn't completely describe all conceivable actions by the government and firms as time unfolds. For example, the definition is silent about how the government would respond if firms, for some unspecified reason, were to choose to deviate from the competitive equilibrium associated with the Ramsey plan, possibly prompting violation of government budget balance. This is an example of the issues raised by *[Bas05]*, who identifies a class of government policy problems whose proper formulation requires supplying a complete and coherent description of all actors' behavior across all possible histories. Implicitly, we are assuming that a more complete description of a government strategy could be specified that (a) agrees with ours along the Ramsey outcome, and (b) suffices uniquely to implement the Ramsey plan by deterring firms from taking actions that deviate from the Ramsey outcome path.

Computing a Ramsey Plan

The planner chooses $\{u_t\}_{t=0}^{\infty}, \{\tau_t\}_{t=1}^{\infty}$ to maximize (7.49) subject to (7.45), (7.46), and (7.48)

To formulate this problem as a Lagrangian, attach a Lagrange multiplier μ to the budget constraint (7.48)

Then the planner chooses $\{u_t\}_{t=0}^{\infty}, \{\tau_t\}_{t=1}^{\infty}$ to maximize and the Lagrange multiplier μ to minimize

$$\sum_{t=0}^{\infty} \beta^t (A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2) + \mu \left[\sum_{t=0}^{\infty} \beta^t \tau_t Q_t - G_0 - \tau_0 Q_0 \right] \tag{7.50}$$

subject to and (7.45) and (7.46)

The Ramsey problem is a special case of the linear quadratic dynamic Stackelberg problem analyzed in *this lecture*

The key implementability conditions are (7.45) for $t \geq 0$

Holding fixed μ and G_0, the Lagrangian for the planning problem can be abbreviated as

$$\max_{\{u_t, \tau_{t+1}\}} \sum_{t=0}^{\infty} \beta^t \left\{ A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 + \mu \tau_t Q_t \right\}$$

Define

$$z_t := \begin{bmatrix} 1 \\ Q_t \\ \tau_t \end{bmatrix} \quad \text{and} \quad y_t := \begin{bmatrix} z_t \\ u_t \end{bmatrix} = \begin{bmatrix} 1 \\ Q_t \\ \tau_t \\ u_t \end{bmatrix}$$

Here the elements of z_t are natural state variables and u_t is a forward looking variable that we treat as a state variable for $t \geq 1$

But u_0 is a choice variable for the Ramsey planner.

We include τ_t as a state variable for bookkeeping purposes: it helps to map the problem into a linear regulator problem with no cross products between states and controls

However, it will be a redundant state variable in the sense that the optimal tax τ_{t+1} will not depend on τ_t

The government chooses τ_{t+1} at time t as a function of the time t state

Thus, we can rewrite the Ramsey problem as

$$\max_{\{y_t, \tau_{t+1}\}} -\sum_{t=0}^{\infty} \beta^t y_t' R y_t \tag{7.51}$$

subject to z_0 given and the law of motion

$$y_{t+1} = A y_t + B \tau_{t+1} \tag{7.52}$$

where

$$R = \begin{bmatrix} 0 & -\frac{A_0}{2} & 0 & 0 \\ -\frac{A_0}{2} & \frac{A_1}{2} & \frac{-\mu}{2} & 0 \\ 0 & \frac{-\mu}{2} & 0 & 0 \\ 0 & 0 & 0 & \frac{d}{2} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ -\frac{A_0}{d} & \frac{A_1}{d} & 0 & \frac{A_1}{d} + \frac{1}{\beta} \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \frac{1}{d} \end{bmatrix}$$

Two Subproblems

Working backwards, we first present the Bellman equation for the value function that takes both z_t and u_t as given. Then we present a value function that takes only z_0 as given and is the indirect utility function that arises from choosing u_0 optimally.

Let $v(Q_t, \tau_t, u_t)$ be the optimum value function for the time $t \geq 1$ government administrator facing state Q_t, τ_t, u_t.

Let $w(Q_0)$ be the value of the Ramsey plan starting from Q_0

Subproblem 1

Here the Bellman equation is

$$v(Q_t, \tau_t, u_t) = \max_{\tau_{t+1}} \left\{ A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 + \mu \tau_t Q_t + \beta v(Q_{t+1}, \tau_{t+1}, u_{t+1}) \right\}$$

where the maximization is subject to the constraints

$$Q_{t+1} = Q_t + u_t$$

and

$$u_{t+1} = -\frac{A_0}{d} + \frac{A_1}{d} Q_t + \frac{A_1}{d} + \frac{1}{\beta} u_t + \frac{1}{d} \tau_{t+1}$$

Here we regard u_t as a state

Subproblem 2

The subproblem 2 Bellman equation is

$$w(z_0) = \max_{u_0} v(Q_0, 0, u_0)$$

Details

Define the state vector to be

$$y_t = \begin{bmatrix} 1 \\ Q_t \\ \tau_t \\ u_t \end{bmatrix} = \begin{bmatrix} z_t \\ u_t \end{bmatrix},$$

where $z_t = \begin{bmatrix} 1 & Q_t & \tau_t \end{bmatrix}'$ are authentic state variables and u_t is a variable whose time 0 value is a 'jump' variable but whose values for dates $t \geq 1$ will become state variables that encode history dependence in the Ramsey plan

$$v(y_t) = \max_{\tau_{t+1}} \left\{ -y_t' R y_t + \beta v(y_{t+1}) \right\} \tag{7.53}$$

where the maximization is subject to the constraint

$$y_{t+1} = A y_t + B \tau_{t+1}$$

and where

$$R = \begin{bmatrix} 0 & -\frac{A_0}{2} & 0 & 0 \\ -\frac{A_0}{2} & \frac{A_1}{2} & \frac{-\mu}{2} & 0 \\ 0 & \frac{-\mu}{2} & 0 & 0 \\ 0 & 0 & 0 & \frac{d}{2} \end{bmatrix}, \; A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ -\frac{A_0}{d} & \frac{A_1}{d} & 0 & \frac{A_1}{d} + \frac{1}{\beta} \end{bmatrix}, \text{ and } B = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \frac{1}{d} \end{bmatrix}.$$

Functional equation (7.53) has solution

$$v(y_t) = -y_t' P y_t$$

where

- P solves the algebraic matrix Riccati equation $P = R + \beta A' P A - \beta A' P B (B' P B)^{-1} B' P A$
- the optimal policy function is given by $\tau_{t+1} = -F y_t$ for $F = (B' P B)^{-1} B' P A$

Now we turn to subproblem 1.

Evidently the optimal choice of u_0 satisfies $\frac{\partial v}{\partial u_0} = 0$

If we partition P as

$$P = \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix}$$

then we have

$$0 = \frac{\partial}{\partial u_0}\left(z_0' P_{11} z_0 + z_0' P_{12} u_0 + u_0' P_{21} z_0 + u_0' P_{22} u_0\right) = P_{12}' z_0 + P_{21} z_0 + 2 P_{22} u_0$$

which implies

$$u_0 = -P_{22}^{-1} P_{21} z_0 \tag{7.54}$$

Thus, the Ramsey plan is

$$\tau_{t+1} = -F \begin{bmatrix} z_t \\ u_t \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} z_{t+1} \\ u_{t+1} \end{bmatrix} = (A - BF) \begin{bmatrix} z_t \\ u_t \end{bmatrix}$$

with initial state $\begin{bmatrix} z_0 & -P_{22}^{-1} P_{21} z_0 \end{bmatrix}'$

Recursive Representation

An outcome of the preceding results is that the Ramsey plan can be represented recursively as the choice of an initial marginal utility (or rate of growth of output) according to a function

$$u_0 = \upsilon(Q_0 | \mu) \tag{7.55}$$

that obeys (7.54) and the following updating equations for $t \geq 0$:

$$\tau_{t+1} = \tau(Q_t, u_t | \mu) \tag{7.56}$$

$$Q_{t+1} = Q_t + u_t \tag{7.57}$$

$$u_{t+1} = u(Q_t, u_t | \mu) \tag{7.58}$$

We have conditioned the functions υ, τ, and u by μ to emphasize how the dependence of F on G_0 appears indirectly through the Lagrange multiplier μ

An Example Calculation

We'll discuss how to compute μ *below* but first consider the following numerical example

We take the parameter set $[A_0, A_1, d, \beta, Q_0] = [100, .05, .2, .95, 100]$ and compute the Ramsey plan with the following piece of code

```
import numpy as np
from quantecon import LQ
from quantecon.matrix_eqn import solve_discrete_lyapunov
from scipy.optimize import root
```

```
def computeG(A0, A1, d, Q0, tau0, beta, mu):
    """
    Compute government income given mu and return tax revenues and
    policy matrixes for the planner.

    Parameters
    ----------
    A0 : float
        A constant parameter for the inverse demand function
    A1 : float
        A constant parameter for the inverse demand function
    d : float
        A constant parameter for quadratic adjustment cost of production
    Q0 : float
        An initial condition for production
    tau0 : float
        An initial condition for taxes
    beta : float
        A constant parameter for discounting
    mu : float
        Lagrange multiplier

    Returns
    -------
    T0 : array(float)
        Present discounted value of government spending
    A : array(float)
        One of the transition matrices for the states
    B : array(float)
        Another transition matrix for the states
    F : array(float)
        Policy rule matrix
    P : array(float)
        Value function matrix
    """
    # Create Matrices for solving Ramsey problem
    R = np.array([[0, -A0/2, 0, 0],
                  [-A0/2, A1/2, -mu/2, 0],
                  [0, -mu/2, 0, 0],
                  [0, 0, 0, d/2]])

    A = np.array([[1, 0, 0, 0],
                  [0, 1, 0, 1],
                  [0, 0, 0, 0],
                  [-A0/d, A1/d, 0, A1/d+1/beta]])

    B = np.array([0, 0, 1, 1/d]).reshape(-1, 1)

    Q = 0

    # Use LQ to solve the Ramsey Problem.
```

```
    lq = LQ(Q, -R, A, B, beta=beta)
    P, F, d = lq.stationary_values()

    # Need y_0 to compute government tax revenue.
    P21 = P[3, :3]
    P22 = P[3, 3]
    z0 = np.array([1, Q0, tau0]).reshape(-1, 1)
    u0 = -P22**(-1) * P21.dot(z0)
    y0 = np.vstack([z0, u0])

    # Define A_F and S matricies
    AF = A - B.dot(F)
    S = np.array([0, 1, 0, 0]).reshape(-1, 1).dot(np.array([[0, 0, 1, 0]]))

    # Solves equation (25)
    temp = beta * AF.T.dot(S).dot(AF)
    Omega = solve_discrete_lyapunov(np.sqrt(beta) * AF.T, temp)
    T0 = y0.T.dot(Omega).dot(y0)

    return T0, A, B, F, P


# == Primitives == #
T    = 20
A0   = 100.0
A1   = 0.05
d    = 0.20
beta = 0.95

# == Initial conditions == #
mu0  = 0.0025
Q0   = 1000.0
tau0 = 0.0


def gg(mu):
    """
    Computes the tax revenues for the government given Lagrangian
    multiplier mu.
    """
    return computeG(A0, A1, d, Q0, tau0, beta, mu)

# == Solve the Ramsey problem and associated government revenue == #
G0, A, B, F, P = gg(mu0)

# == Compute the optimal u0 == #
P21 = P[3, :3]
P22 = P[3, 3]
z0 = np.array([1, Q0, tau0]).reshape(-1, 1)
u0 = -P22**(-1) * P21.dot(z0)


# == Initialize vectors == #
```

```
y = np.zeros((4, T))
uhat        = np.zeros(T)
uhatdif     = np.zeros(T)
tauhat      = np.zeros(T)
tauhatdif   = np.zeros(T-1)
mu          = np.zeros(T)
G           = np.zeros(T)
GPay        = np.zeros(T)

# == Initial conditions == #
G[0] = G0
mu[0] = mu0
uhatdif[0] = 0
uhat[0] = u0
y[:, 0] = np.vstack([z0, u0]).flatten()

for t in range(1, T):
    # Iterate government policy
    y[:, t] = (A-B.dot(F)).dot(y[:, t-1])

    # update G
    G[t] = (G[t-1] - beta*y[1, t]*y[2, t])/beta
    GPay[t] = beta*y[1, t]*y[2, t]

    # Compute the mu if the government were able to reset its plan
    # ff is the tax revenues the government would receive if they reset the
    # plan with Lagrange multiplier mu minus current G

    ff = lambda mu: (gg(mu)[0]-G[t]).flatten()

    # find ff = 0
    mu[t] = root(ff, mu[t-1]).x
    temp, Atemp, Btemp, Ftemp, Ptemp = gg(mu[t])

    # Compute alternative decisions
    P21temp = Ptemp[3, :3]
    P22temp = P[3, 3]
    uhat[t] = -P22temp**(-1)*P21temp.dot(y[:3, t])

    yhat = (Atemp-Btemp.dot(Ftemp)).dot(np.hstack([y[0:3, t-1], uhat[t-1]]))
    tauhat[t] = yhat[3]
    tauhatdif[t-1] = tauhat[t]-y[3, t]
    uhatdif[t] = uhat[t]-y[3, t]


if __name__ == '__main__':
    print("1 Q tau u")
    print(y)
    print("-F")
    print(-F)
```

The program can also be found in

It computes a number of sequences besides the Ramsey plan, some of which have already been discussed, while others will be described below

The next figure uses the program to compute and show the Ramsey plan for τ and the Ramsey outcome for (Q_t, u_t)

From top to bottom, the panels show Q_t, τ_t and $u_t := Q_{t+1} - Q_t$ over $t = 0, \ldots, 15$

The optimal decision rule is[4]

$$\tau_{t+1} = -248.0624 - 0.1242 Q_t - 0.3347 u_t \tag{7.59}$$

Notice how the Ramsey plan calls for a high tax at $t = 1$ followed by a perpetual stream of lower taxes

Taxing heavily at first, less later expresses time-inconsistency of the optimal plan for $\{\tau_{t+1}\}_{t=0}^{\infty}$

We'll characterize this formally after first discussing how to compute μ.

Computing μ

Define the selector vectors $e_\tau = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}'$ and $e_Q = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}'$ and express $\tau_t = e_\tau' y_t$ and $Q_t = e_Q' y_t$

[4] As promised, τ_t does not appear in the Ramsey planner's decision rule for τ_{t+1}.

Evidently $Q_t \tau_t = y_t' e_Q e_\tau' y_t = y_t' S y_t$ where $S := e_Q e_\tau'$

We want to compute

$$T_0 = \sum_{t=1}^{\infty} \beta^t \tau_t Q_t = \tau_1 Q_1 + \beta T_1$$

where $T_1 = \sum_{t=2}^{\infty} \beta^{t-1} Q_t \tau_t$

The present values T_0 and T_1 are connected by

$$T_0 = \beta y_0' A_F' S A_F y_0 + \beta T_1$$

Guess a solution that takes the form $T_t = y_t' \Omega y_t$, then find an Ω that satisfies

$$\Omega = \beta A_F' S A_F + \beta A_F' \Omega A_F \tag{7.60}$$

Equation (7.60) is a discrete Lyapunov equation that can be solved for Ω using QuantEcon's solve_discrete_lyapunov function

The matrix F and therefore the matrix $A_F = A - BF$ depend on μ

To find a μ that guarantees that $T_0 = G_0$ we proceed as follows:

1. Guess an initial μ, compute a tentative Ramsey plan and the implied $T_0 = y_0' \Omega(\mu) y_0$
2. If $T_0 > G_0$, lower μ; otherwise, raise μ
3. Continue iterating on step 3 until $T_0 = G_0$

Time Inconsistency

Recall that the Ramsey planner chooses $\{u_t\}_{t=0}^{\infty}, \{\tau_t\}_{t=1}^{\infty}$ to maximize

$$\sum_{t=0}^{\infty} \beta^t \left\{ A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 \right\}$$

subject to (7.45), (7.46), and (7.48)

We express the outcome that a Ramsey plan is time-inconsistent the following way

Proposition. A continuation of a Ramsey plan is not a Ramsey plan

Let

$$w(Q_0, u_0 | \mu_0) = \sum_{t=0}^{\infty} \beta^t \left\{ A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 \right\} \tag{7.61}$$

where

- $\{Q_t, u_t\}_{t=0}^{\infty}$ are evaluated under the Ramsey plan whose recursive representation is given by (7.56), (7.57), (7.58)
- μ_0 is the value of the Lagrange multiplier that assures budget balance, computed as described *above*

Evidently, these continuation values satisfy the recursion

$$w(Q_t, u_t|\mu_0) = A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 + \beta w(Q_{t+1}, u_{t+1}|\mu_0) \tag{7.62}$$

for all $t \geq 0$, where $Q_{t+1} = Q_t + u_t$

Under the timing protocol affiliated with the Ramsey plan, the planner is committed to the outcome of iterations on (7.56), (7.57), (7.58)

In particular, when time t comes, the Ramsey planner is committed to the value of u_t implied by the Ramsey plan and receives continuation value $w(Q_t, u_t, \mu_0)$

That the Ramsey plan is time-inconsistent can be seen by subjecting it to the following 'revolutionary' test

First, define continuation revenues G_t that the government raises along the original Ramsey outcome by

$$G_t = \beta^{-t}(G_0 - \sum_{s=1}^{t} \beta^s \tau_s Q_s) \tag{7.63}$$

where $\{\tau_t, Q_t\}_{t=0}^{\infty}$ is the original Ramsey outcome[5]

Then at time $t \geq 1$,

1. take (Q_t, G_t) inherited from the original Ramsey plan as initial conditions
2. invite a brand new Ramsey planner to compute a new Ramsey plan, solving for a new u_t, to be called $\check{u}_t$, and for a new μ, to be called $\check{\mu}_t$

The revised Lagrange multiplier $\check{\mu}_t$ is chosen so that, under the new Ramsey plan, the government is able to raise enough continuation revenues G_t given by (7.63)

Would this new Ramsey plan be a continuation of the original plan?

The answer is no because along a Ramsey plan, for $t \geq 1$, in general it is true that

$$w\big(Q_t, v(Q_t|\check{\mu})|\check{\mu}\big) > w(Q_t, u_t|\mu_0) \tag{7.64}$$

Inequality (7.64) expresses a continuation Ramsey planner's incentive to deviate from a time 0 Ramsey plan by

1. resetting u_t according to (7.55)
2. adjusting the Lagrange multiplier on the continuation appropriately to account for tax revenues already collected[6]

Inequality (7.64) expresses the time-inconsistency of a Ramsey plan

[5] The continuation revenues G_t are the time t present value of revenues that must be raised to satisfy the original time 0 government intertemporal budget constraint, taking into account the revenues already raised from $s = 1, \ldots, t$ under the original Ramsey plan.

[6] For example, let the Ramsey plan yield time 1 revenues $Q_1 \tau_1$. Then at time 1, a continuation Ramsey planner would want to raise continuation revenues, expressed in units of time 1 goods, of $\tilde{G}_1 := \frac{G - \beta Q_1 \tau_1}{\beta}$. To finance the remainder revenues, the continuation Ramsey planner would find a continuation Lagrange multiplier μ by applying the three-step procedure from the previous section to revenue requirements $\tilde{G}_1$.

A Simulation

To bring out the time inconsistency of the Ramsey plan, we compare

- the time t values of τ_{t+1} under the original Ramsey plan with
- the value $\check{\tau}_{t+1}$ associated with a new Ramsey plan begun at time t with initial conditions (Q_t, G_t) generated by following the *original* Ramsey plan

Here again $G_t := \beta^{-t}(G_0 - \sum_{s=1}^{t} \beta^s \tau_s Q_s)$

The difference $\Delta\tau_t := \check{\tau}_t - \tau_t$ is shown in the top panel of the following figure

In the second panel we compare the time t outcome for u_t under the original Ramsey plan with the time t value of this new Ramsey problem starting from (Q_t, G_t)

To compute u_t under the new Ramsey plan, we use the following version of formula (7.54):

$$\check{u}_t = -P_{22}^{-1}(\check{\mu}_t)P_{21}(\check{\mu}_t)z_t$$

Here z_t is evaluated along the Ramsey outcome path, where we have included $\check{\mu}_t$ to emphasize the dependence of P on the Lagrange multiplier μ_0[7]

[7] It can be verified that this formula puts non-zero weight only on the components 1 and Q_t of z_t.

To compute u_t along the Ramsey path, we just iterate the recursion starting (7.58) from the initial Q_0 with u_0 being given by formula (7.54)

Thus the second panel indicates how far the reinitialized value $\check{u}_t$ value departs from the time t outcome along the Ramsey plan

Note that the restarted plan raises the time $t+1$ tax and consequently lowers the time t value of u_t

Associated with the new Ramsey plan at t is a value of the Lagrange multiplier on the continuation government budget constraint

This is the third panel of the figure

The fourth panel plots the required continuation revenues G_t implied by the original Ramsey plan

These figures help us understand the time inconsistency of the Ramsey plan

Further Intuition

One feature to note is the large difference between $\check{\tau}_{t+1}$ and τ_{t+1} in the top panel of the figure

If the government is able to reset to a new Ramsey plan at time t, it chooses a significantly higher tax rate than if it were required to maintain the original Ramsey plan

The intuition here is that the government is required to finance a given present value of expenditures with distorting taxes τ

The quadratic adjustment costs prevent firms from reacting strongly to variations in the tax rate for next period, which tilts a time t Ramsey planner toward using time $t+1$ taxes

As was noted before, this is evident in *the first figure*, where the government taxes the next period heavily and then falls back to a constant tax from then on

This can also been seen in the third panel of *the second figure*, where the government pays off a significant portion of the debt using the first period tax rate

The similarities between the graphs in the last two panels of *the second figure* reveals that there is a one-to-one mapping between G and μ

The Ramsey plan can then only be time consistent if G_t remains constant over time, which will not be true in general

Credible Policy

We express the theme of this section in the following: In general, a continuation of a Ramsey plan is not a Ramsey plan

This is sometimes summarized by saying that a Ramsey plan is not *credible*

On the other hand, a continuation of a credible plan is a credible plan

The literature on a credible public policy (*[CK90]* and *[Sto89]*) arranges strategies and incentives so that public policies can be implemented by a *sequence* of government decision makers instead of a single Ramsey planner who chooses an entire sequence of history-dependent actions once and for all at time $t = 0$

Here we confine ourselves to sketching how recursive methods can be used to characterize credible policies in our model

A key reference on these topics is *[Cha98]*

A credibility problem arises because we assume that the timing of decisions differs from those for a Ramsey problem

A **sequential timing protocol** is a protocol such that

1. At each $t \geq 0$, given Q_t and expectations about a continuation tax policy $\{\tau_{s+1}\}_{s=t}^{\infty}$ and a continuation price sequence $\{p_{s+1}\}_{s=t}^{\infty}$, the representative firm chooses u_t
2. At each t, given (Q_t, u_t), a government chooses τ_{t+1}

Item (2) captures that taxes are now set sequentially, the time $t+1$ tax being set *after* the government has observed u_t

Of course, the representative firm sets u_t in light of its expectations of how the government will ultimately choose to set future taxes

A credible tax plan $\{\tau_{s+1}\}_{s=t}^{\infty}$

- is anticipated by the representative firm, and
- is one that a time t government chooses to confirm

We use the following recursion, closely related to but different from (7.62), to define the continuation value function for the government:

$$J_t = A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 + \beta J_{t+1}(\tau_{t+1}, G_{t+1}) \tag{7.65}$$

This differs from (7.62) because

- continuation values are now allowed to depend explicitly on values of the choice τ_{t+1}, and
- continuation government revenue to be raised G_{t+1} need not be ones called for by the prevailing government policy

Thus, deviations from that policy are allowed, an alteration that recognizes that τ_t is chosen sequentially

Express the government budget constraint as requiring that G_0 solves the difference equation

$$G_t = \beta \tau_{t+1} Q_{t+1} + \beta G_{t+1}, \quad t \geq 0 \tag{7.66}$$

subject to the terminal condition $\lim_{t \to +\infty} \beta^t G_t = 0$

Because the government is choosing sequentially, it is convenient to

- take G_t as a state variable at t and
- to regard the time t government as choosing (τ_{t+1}, G_{t+1}) subject to constraint (7.66)

To express the notion of a credible government plan concisely, we expand the strategy space by also adding J_t itself as a state variable and allowing policies to take the following recursive forms[8]

[8] This choice is the key to what *[LS12]* call 'dynamic programming squared'.

Regard J_0 as an a discounted present value promised to the Ramsey planner and take it as an initial condition.

Then after choosing u_0 according to

$$u_0 = \upsilon(Q_0, G_0, J_0), \tag{7.67}$$

choose subsequent taxes, outputs, *and* continuation values according to recursions that can be represented as

$$\hat{\tau}_{t+1} = \tau(Q_t, u_t, G_t, J_t) \tag{7.68}$$

$$u_{t+1} = \xi(Q_t, u_t, G_t, J_t, \tau_{t+1}) \tag{7.69}$$

$$G_{t+1} = \beta^{-1} G_t - \tau_{t+1} Q_{t+1} \tag{7.70}$$

$$J_{t+1}(\tau_{t+1}, G_{t+1}) = \nu(Q_t, u_t, G_{t+1}, J_t, \tau_{t+1}) \tag{7.71}$$

Here

- $\hat{\tau}_{t+1}$ is the time $t+1$ government action called for by the plan, while
- τ_{t+1} is possibly some one-time deviation that the time $t+1$ government contemplates and
- G_{t+1} is the associated continuation tax collections

The plan is said to be **credible** if, for each t and each state (Q_t, u_t, G_t, J_t), the plan satisfies the incentive constraint

$$\begin{aligned} J_t &= A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 + \beta J_{t+1}(\hat{\tau}_{t+1}, \hat{G}_{t+1}) \\ &\geq A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 + \beta J_{t+1}(\tau_{t+1}, G_{t+1}) \end{aligned} \tag{7.72}$$

for all tax rates $\tau_{t+1} \in \mathbb{R}$ available to the government

Here $\hat{G}_{t+1} = \frac{G_t - \hat{\tau}_{t+1} Q_{t+1}}{\beta}$

- Inequality expresses that continuation values adjust to deviations in ways that discourage the government from deviating from the prescribed $\hat{\tau}_{t+1}$
- Inequality (7.72) indicates that *two* continuation values J_{t+1} contribute to sustaining time t promised value J_t
  - $J_{t+1}(\hat{\tau}_{t+1}, \hat{G}_{t+1})$ is the continuation value when the government chooses to confirm the private sector's expectation, formed according to the decision rule (7.68)[9]

[9] Note the double role played by (7.68): as decision rule for the government and as the private sector's rule for forecasting government actions.

- $J_{t+1}(\tau_{t+1}, G_{t+1})$ tells the continuation consequences should the government disappoint the private sector's expectations

The internal structure of a credible plan deters deviations from it

That (7.72) maps *two* continuation values $J_{t+1}(\tau_{t+1}, G_{t+1})$ and $J_{t+1}(\hat{\tau}_{t+1}, \hat{G}_{t+1})$ into one promised value J_t reflects how a credible plan arranges a system of private sector expectations that induces the government to choose to confirm them

Chang *[Cha98]* builds on how inequality (7.72) maps two continuation values into one

Remark Let $\mathcal{J}$ be the set of values associated with credible plans

Every value $J \in \mathcal{J}$ can be attained by a credible plan that has a recursive representation of form form (7.68), (7.69), (7.70)

The set of values can be computed as the largest fixed point of an operator that maps sets of candidate values into sets of values

Given a value within this set, it is possible to construct a government strategy of the recursive form (7.68), (7.69), (7.70) that attains that value

In many cases, there is a **set** of values and associated credible plans

In those cases where the Ramsey outcome is credible, a multiplicity of credible plans is a key part of the story because, as we have seen earlier, a continuation of a Ramsey plan is not a Ramsey plan

For it to be credible, a Ramsey outcome must be supported by a worse outcome associated with another plan, the prospect of reversion to which sustains the Ramsey outcome

Concluding remarks

The term 'optimal policy', which pervades an important applied monetary economics literature, means different things under different timing protocols

Under the 'static' Ramsey timing protocol (i.e., choose a sequence once-and-for-all), we obtain a unique plan

Here the phrase 'optimal policy' seems to fit well, since the Ramsey planner optimally reaps early benefits from influencing the private sector's beliefs about the government's later actions

When we adopt the sequential timing protocol associated with credible public policies, 'optimal policy' is a more ambiguous description

There is a multiplicity of credible plans

True, the theory explains how it is optimal for the government to confirm the private sector's expectations about its actions along a credible plan

But some credible plans have very bad outcomes

These bad outcomes are central to the theory because it is the presence of bad credible plans that makes possible better ones by sustaining the low continuation values that appear in the second line of incentive constraint (7.72)

Recently, many have taken for granted that 'optimal policy' means 'follow the Ramsey plan'[10]

In pursuit of more attractive ways to describe a Ramsey plan when policy making is in practice done sequentially, some writers have repackaged a Ramsey plan in the following way

- Take a Ramsey *outcome* - a sequence of endogenous variables under a Ramsey plan - and reinterpret it (or perhaps only a subset of its variables) as a *target path* of relationships among outcome variables to be assigned to a sequence of policy makers[11]
- If appropriate (infinite dimensional) invertibility conditions are satisfied, it can happen that following the Ramsey plan is the *only* way to hit the target path[12]
- The spirit of this work is to say, "in a democracy we are obliged to live with the sequential timing protocol, so let's constrain policy makers' objectives in ways that will force them to follow a Ramsey plan in spite of their benevolence"[13]
- By this slight of hand, we acquire a theory of an *optimal outcome target path*

This 'invertibility' argument leaves open two important loose ends:

1. implementation, and
2. time consistency

As for (1), repackaging a Ramsey plan (or the tail of a Ramsey plan) as a target outcome sequence does not confront the delicate issue of *how* that target path is to be implemented[14]

As for (2), it is an interesting question whether the 'invertibility' logic can repackage and conceal a Ramsey plan well enough to make policy makers forget or ignore the benevolent intentions that give rise to the time inconsistency of a Ramsey plan in the first place

To attain such an optimal output path, policy makers must forget their benevolent intentions because there will inevitably occur temptations to deviate from that target path, and the implied relationship among variables like inflation, output, and interest rates along it

Remark The continuation of such an optimal target path is not an optimal target path

Optimal Taxation with State-Contingent Debt

Contents

- *Optimal Taxation with State-Contingent Debt*
  - *Overview*
  - *A competitive equilibrium with distorting taxes*

[10] It is possible to read *[Woo03]* and *[GW10]* as making some carefully qualified statements of this type. Some of the qualifications can be interpreted as advice 'eventually' to follow a tail of a Ramsey plan.

[11] In our model, the Ramsey outcome would be a path $(\vec{p}, \vec{Q})$.

[12] See *[GW10]*.

[13] Sometimes the analysis is framed in terms of following the Ramsey plan only from some future date T onwards.

[14] See *[Bas05]* and *[ACK10]*.

- *Recursive formulation of the Ramsey problem*
- *Examples*
- *Implementation*

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

The model features

- a government that must finance an exogenous stream of government expenditures with either
  - a flat rate tax on labor, or
  - purchases and sales of some a full array of Arrow state contingent securities
- a representative consumer who values consumption and leisure
- a linear production function mapping labor into a single good
- a Ramsey planner who at time $t = 0$ chooses a plan for taxes and trades of Arrow securities for all $t \geq 0$

After first presenting the model in a space of sequences, we shall represent it recursively in terms of two Bellman equations formulated along lines that we encountered in *Dynamic Stackelberg models*

As in *Dynamic Stackelberg models*, to apply dynamic programming we shall define the state vector artfully

In particular, we shall include forward-looking variables that summarize optimal responses of private agents to a Ramsey plan

See *Optimal taxation* for an analysis within a linear-quadratic setting

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 ℓ_t and labor n_t:

$$n_t(s^t) + \ell_t(s^t) = 1 \tag{7.73}$$

Output equals $n_t(s^t)$ and can be divided between $c_t(s^t)$ and $g_t(s^t)$

$$c_t(s^t) + g_t(s^t) = n_t(s^t) \tag{7.74}$$

A representative household's preferences over $\{c_t(s^t), \ell_t(s^t)\}_{t=0}^{\infty}$ are ordered by

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)] \tag{7.75}$$

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 $p_t(s_{t+1}|s^t)$

The government issues one-period Arrow securities each period

The government has a sequence of budget constraints whose time $t \geq 0$ component is

$$g_t(s^t) = \tau_t(s^t) n_t(s^t) + \sum_{s_{t+1}} p_t(s_{t+1}|s^t) b_{t+1}(s_{t+1}|s^t) - b_t(s_t|s^{t-1}) \tag{7.76}$$

where

- $p_t(s_{t+1}|s^t)$ 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.

Here $p_t(s_{t+1}|s^t)$ is the price of an Arrow security

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

$$c_t(s^t) + \sum_{s_{t+1}} p_t(s_{t+1}|s^t) b_{t+1}(s_{t+1}|s^t) = \left[1 - \tau_t(s^t)\right] n_t(s^t) + b_t(s_t|s^{t-1}) \quad \forall t \geq 0. \tag{7.77}$$

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 (7.74) at all t, s^t

A **price system** is a sequence of Arrow securities prices $\{p_t(s_{t+1}|s^t)\}_{t=0}^{\infty}$

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 (7.75) subject to (7.77) and (7.73) 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 household's optimization problem.
- Given the allocation, government policy, and price system, the government's budget constraint is satisfied for all t, s^t

Note: There is a large number of competitive equilibria with distorting taxes, indexed by different government policies.

The **Ramsey problem** or **optimal taxation problem** is to choose a competitive equilibrium with distorting taxes that maximizes (7.75)

Arrow-Debreu version of price system

We find it convenient sometimes to work with the Arrow-Debreu price system implied by a sequence of Arrow securities prices

Let $q_t^0(s^t)$ be the price at time 0, measured in time 0 consumption goods, of one unit of consumption at time t, history s^t

The following recursion relates Arrow-Debreu prices $\{q_t^0(s^t)\}_{t=0}^\infty$ to the Arrow securities prices $\{p_t(s_{t+1}|s^t)\}_{t=0}^\infty$

$$q_{t+1}^0(s^{t+1}) = p_t(s_{t+1}|s^t)q_t^0(s^t) \tag{7.78}$$

subject to $q_0^0(s^0) = 1$

These 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 optimum 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 the first-order conditions of the household's problem and solve these conditions for $\{q_t^0(s^t), \tau_t(s^t)\}_{t=0}^\infty$ as functions of the allocation $\{c_t(s^t), n_t(s^t)\}_{t=0}^\infty$.
2. Substitute these expressions for taxes and prices in terms of the allocation into the household's present-value budget constraint
   - This intertemporal constraint involves only the allocation and is regarded as an *implementability constraint*.
3. Find the allocation that maximizes the utility of the representative consumer (7.75) subject to the feasibility constraints (7.73) and (7.74) 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 (7.77) into another, we can obtain the household's present-value budget constraint:

$$\sum_{t=0}^{\infty} \sum_{s^t} q_t^0(s^t) c_t(s^t) = \sum_{t=0}^{\infty} \sum_{s^t} q_t^0(s^t)[1-\tau_t(s^t)] w_t(s^t) n_t(s^t) + b_0 \tag{7.79}$$

Here $\{q_t^0(s^t)\}_{t=1}^{\infty}$ can be interpreted as a time 0 Arrow-Debreu price system

To approach the Ramsey problem, we study the household's optimization problem

First-order conditions for the household's problem for $\ell_t(s^t)$ and $b_t(s_{t+1}|s^t)$, respectively, imply

$$(1-\tau_t(s^t)) = \frac{u_l(s^t)}{u_c(s^t)} \tag{7.80}$$

and

$$p_{t+1}(s_{t+1}|s^t) = \beta \pi(s_{t+1}|s^t) \left(\frac{u_c(s^{t+1})}{u_c(s^t)} \right) \tag{7.81}$$

where $\pi(s_{t+1}|s^t)$ is the probability distribution of s_{t+1} conditional on history s^t

Equation (7.81) implies that the Arrow-Debreu price system satisfies

$$q_t^0(s^t) = \beta^t \pi_t(s^t) \frac{u_c(s^t)}{u_c(s^0)} \tag{7.82}$$

Use the first-order conditions (7.80) and (7.81) to eliminate taxes and prices from (7.79) to derive the *implementability condition*

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t)[u_c(s^t)c_t(s^t) - u_\ell(s^t)n_t(s^t)] - u_c(s^0)b_0 = 0. \tag{7.83}$$

The **Ramsey problem** is to choose a feasible allocation that maximizes

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), 1-n_t(s^t)] \tag{7.84}$$

subject to (7.83)

Solution details

First define a "pseudo utility function"

$$V\left[c_t(s^t), n_t(s^t), \Phi\right] = u[c_t(s^t), 1-n_t(s^t)] + \Phi\left[u_c(s^t)c_t(s^t) - u_\ell(s^t)n_t(s^t)\right], \tag{7.85}$$

where Φ is a Lagrange multiplier on the implementability condition (7.79).

Next form the Lagrangian

$$J = \sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) \Big\{ V[c_t(s^t), n_t(s^t), \Phi] + \theta_t(s^t) \Big[n_t(s^t) - c_t(s^t) - g_t(s_t) \Big] \Big\} - \Phi u_c(0) b_0 \tag{7.86}$$

where $\{\theta_t(s^t); \forall s^t\}_{t\geq 0}$ is a sequence of Lagrange multipliers on the feasible conditions (7.74)

For given initial government debt b_0, we want to maximize J with respect to $\{c_t(s^t), n_t(s^t); \forall s^t\}_{t\geq 0}$ and to minimize with respect to $\{\theta(s^t); \forall s^t\}_{t\geq 0}$.

The first-order conditions for the Ramsey problem for periods $t \geq 1$ and $t = 0$, respectively, are

$$\begin{aligned} c_t(s^t)\colon (1+\Phi)u_c(s^t) + \Phi\left[u_{cc}(s^t)c_t(s^t) - u_{\ell c}(s^t)n_t(s^t)\right] - \theta_t(s^t) = 0, \quad t \geq 1, \\ n_t(s^t)\colon -(1+\Phi)u_\ell(s^t) - \Phi\left[u_{c\ell}(s^t)c_t(s^t) - u_{\ell\ell}(s^t)n_t(s^t)\right] + \theta_t(s^t) = 0, \quad t \geq 1 \end{aligned} \tag{7.87}$$

and

$$\begin{aligned} c_0(s^0, b_0)\colon (1+\Phi)u_c(s^0, b_0) + \Phi\left[u_{cc}(s^0, b_0)c_0(s^0, b_0) - u_{\ell c}(s^0, b_0)n_0(s^0, b_0)\right] - \theta_0(s^0, b_0) \\ - \Phi u_{cc}(s^0, b_0)b_0 = 0, \\ n_0(s^0, b_0)\colon -(1+\Phi)u_\ell(s^0, b_0) - \Phi\left[u_{c\ell}(s^0, b_0)c_0(s^0, b_0) - u_{\ell\ell}(s^0, b_0)n_0(s^0, b_0)\right] + \theta_0(s^0, b_0) \\ + \Phi u_{c\ell}(s^0, b_0)b_0 = 0. \end{aligned} \tag{7.88}$$

It is instructive to use first-order conditions (7.87) for $t \geq 1$ to eliminate the multipliers $\theta_t(s^t)$

For convenience, we suppress the time subscript and the index s^t and obtain

$$\begin{aligned} (1+\Phi)u_c(c, 1-c-g) + \Phi\big[cu_{cc}(c, 1-c-g) - (c+g)u_{\ell c}(c, 1-c-g)\big] \\ = (1+\Phi)u_\ell(c, 1-c-g) + \Phi\big[cu_{c\ell}(c, 1-c-g) - (c+g)u_{\ell\ell}(c, 1-c-g)\big], \end{aligned} \tag{7.89}$$

where we have imposed conditions (7.73) and (7.74)

Equation (7.89) is one equation that can be solved to express the unknown c as a function of the exogenous variable g

Notice that a counterpart to $b_t(s_t|s^{t-1})$ does *not* appear in (7.89), 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 (7.88) leads to one equation that can be solved for c_0 and 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 (7.89) 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 preceding that realization of g

The Ramsey allocation for a given Φ

Temporarily take Φ as given.

We shall compute $c_0(s^0, b_0)$ and $n_0(s^0, b_0)$ from the first-order conditions (7.88).

Evidently, for $t \geq 1$, c and n depend on the time t realization of g only.

But for $t = 0$, c and n depend on both g_0 and the government's initial debt b_0.

Thus, while b_0 influences c_0 and n_0, there appears no analogous variable b_t that influences c_t and n_t for $t \geq 1$.

The absence of b_t as a determinant of the Ramsey allocation for $t \geq 1$ and its presence for $t = 0$ is a symptom of the *time-inconsistency* of a Ramsey plan.

Φ has to take a value that assures that the consumer's and the government's budget constraints are both satisfied at a candidate Ramsey allocation and price system associated with that Φ.

Further specialization

At this point, it is useful to specialize the model in the following ways.

We assume that s is governed by a finite state Markov chain with states $s \in [1, \ldots, S]$ and transition matrix Π, where

$$\Pi(s'|s) = \text{Prob}(s_{t+1} = s'|s_t = s).$$

Also, assume that government purchases g are an exact time-invariant function $g(s)$ of s.

We maintain these assumptions throughout the remainder of this lecture.

Determining Φ

We complete the Ramsey plan by computing the Lagrange multiplier Φ on the implementability constraint (7.83)

Government budget balance restricts Φ via the following line of reasoning

The household's first-order conditions imply

$$(1 - \tau_t(s^t)) = \frac{u_l(s^t)}{u_c(s^t)} \tag{7.90}$$

and the implied one-period Arrow securities prices

$$p_{t+1}(s_{t+1}|s^t) = \beta \Pi(s_{t+1}|s_t) \frac{u_c(s^{t+1})}{u_c(s^t)}. \tag{7.91}$$

Substituting from (7.90), (7.91), and the feasibility condition (7.74) into the recursive version (7.77) of the household budget constraint gives

$$\begin{aligned} u_c(s^t)[n_t(s^t) - g_t(s^t)] + \beta \sum_{s_{t+1}} \Pi(s_{t+1}|s_t) u_c(s^{t+1}) b_{t+1}(s_{t+1}|s^t) \\ = u_l(s^t) n_t(s^t) + u_c(s^t) b_t(s_t|s^{t-1}). \end{aligned} \tag{7.92}$$

Define the product $x_t(s^t) = u_c(s^t) b_t(s_t|s^{t-1})$

Notice that $x_t(s^t)$ appears on the right side of while β times the conditional expectation of $x_{t+1}(s^{t+1})$ appears on the left side

Hence the equation shares much of the structure of a simple asset pricing equation with x_t being analogous to the price of the asset at time t.

We learned earlier that for a Ramsey allocation, $c_t(s^t), n_t(s^t)$ and $b_t(s_t|s^{t-1})$, and therefore also $x_t(s^t)$, are each functions of s_t only, being independent of the history s^{t-1} for $t \geq 1$.

That means that we can express equation (7.92) as

$$u_c(s)[n(s) - g(s)] + \beta \sum_{s'} \Pi(s'|s)x'(s') = u_l(s)n(s) + x(s), \tag{7.93}$$

where s' denotes a next period value of s and $x'(s')$ denotes a next period value of x.

Equation (7.93) is easy to solve for $x(s)$ for $s = 1, \ldots, S$.

If we let $\vec{n}, \vec{g}, \vec{x}$ denote $S \times 1$ vectors whose ith elements are the respective n, g, and x values when $s = i$, and let Π be the transition matrix for the Markov state s, then we can express as the matrix equation

$$\vec{u}_c(\vec{n} - \vec{g}) + \beta \Pi \vec{x} = \vec{u}_l \vec{n} + \vec{x}. \tag{7.94}$$

This is a system of S linear equations in the $S \times 1$ vector x, whose solution is

$$\vec{x} = (I - \beta \Pi)^{-1}[\vec{u}_c(\vec{n} - \vec{g}) - \vec{u}_l \vec{n}]. \tag{7.95}$$

In these equations, by $\vec{u}_c \vec{n}$, for example, we mean element-by-element multiplication of the two vectors.

After solving for $\vec{x}$, we can find $b(s_t|s^{t-1})$ in Markov state $s_t = s$ from $b(s) = \frac{x(s)}{u_c(s)}$ or the matrix equation

$$\vec{b} = \frac{\vec{x}}{\vec{u}_c}, \tag{7.96}$$

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 Φ, then use the first-order conditions and the feasibility conditions to compute $c(s_t), n(s_t)$ for $s \in [1, \ldots, S]$ and $c_0(s_0, b_0)$ and $n_0(s_0, b_0)$, given Φ. These are $2(S+1)$ equations in $2(S+1)$ unknowns.
2. Solve the S equations (7.95) for the S elements of $\vec{x}$. These depend on Φ.
3. Find a Φ that satisfies

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + \beta \sum_{s=1}^{S} \Pi(s|s_0) x(s) \tag{7.97}$$

   by gradually raising Φ if the left side of (7.97) exceeds the right side and lowering Φ if the left side is smaller.

4. After computing a Ramsey allocation, recover the flat tax rate on labor from (7.80) and the implied one-period Arrow securities prices from (7.81)

In summary, when g_t is a time invariant function of a Markov state s_t, a Ramsey plan can be constructed by solving $3S + 3$ equations in S components each of $\vec{c}$, $\vec{n}$, and $\vec{x}$ together with n_0, c_0, and Φ.

Time inconsistency

Let $\{\tau_t(s^t)\}_{t=0}^{\infty}$, $\{b_{t+1}(s_{t+1}|s^t)\}_{t=0}^{\infty}$ be a time 0, state s_0 Ramsey plan.

Then $\{\tau_j(s^j)\}_{j=t}^{\infty}$, $\{b_{j+1}(s_{j+1}|s^j)\}_{j=t}^{\infty}$ is a time t, history s^t continuation of a time 0, state s_0 Ramsey plan.

A time t, history s^t Ramsey plan is a Ramsey plan that starts from initial conditions $s^t, b_t(s_t|s^{t-1})$.

A time t, history s^t continuation of a time 0, state 0 Ramsey plan is *not* a time t, history s^t Ramsey plan.

The means that a Ramsey plan is *not time consistent*

Another way to say the same thing is that a Ramsey plan is *time inconsistent*

The reason is that the continuation Ramsey plan takes $u_{ct}b_t(s_t|s^{t-1})$ as given, not $b_t(s_t|s^{t-1})$

We shall discuss this more below

Recursive formulation of the Ramsey problem

$x_t(s^t) = u_c(s^t)b_t(s_t|s^{t-1})$ in equation (7.92) appears to be a purely "forward-looking" variable.

But $x_t(s^t)$ is a also a natural candidate for a state variable in a recursive formulation of the Ramsey problem

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 $t \geq 1$ are obligated to attain.

Continuation Ramsey planners do this by choosing continuation policies that induce the representative household to make choices that imply that $u_c(s^t)b_t(s_t|s^{t-1}) = x_t(s^t)$

A time $t \geq 1$ continuation Ramsey planner 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, not x_0, as a state variable.

Furthermore, the Ramsey planner cares about $(c_0(s_0), \ell_0(s_0))$, while continuation Ramsey planners do not.

The time 0 Ramsey planner hands x_1 as a function of s_1 to a time 1 continuation Ramsey planner.

These lines of delegated authorities and responsibilities across time express the continuation Ramsey planners' obligations to implement their parts of the original Ramsey plan, designed once-and-for-all at time 0.

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

$$V(x, s) = \max_{n, \{x'(s')\}} u(n - g(s), 1 - n) + \beta \sum_{s' \in S} \Pi(s'|s) V(x', s') \tag{7.98}$$

where maximization over n and the S elements of $x'(s')$ is subject to the single implementability constraint for $t \geq 1$

$$x = u_c(n - g(s)) - u_l n + \beta \sum_{s' \in \mathcal{S}} \Pi(s'|s) x'(s') \tag{7.99}$$

Here u_c and u_l are today's values of the marginal utilities.

For each given value of x, s, the continuation Ramsey planner chooses n and an $x'(s')$ for each $s' \in \mathcal{S}$.

Associated with a value function $V(x, s)$ that solves Bellman equation are $S + 1$ time-invariant policy functions

$$\begin{aligned} n_t &= f(x_t, s_t), \quad t \geq 1 \\ x_{t+1}(s_{t+1}) &= h(s_{t+1}; x_t, s_t),\ s_{t+1} \in \mathcal{S},\ t \geq 1. \end{aligned} \tag{7.100}$$

The Ramsey problem

The Bellman equation for the time 0 Ramsey planner is

$$W(b_0, s_0) = \max_{n_0, \{x'(s_1)\}} u(n_0 - g_0, 1 - n_0) + \beta \sum_{s_1 \in \mathcal{S}} \Pi(s_1|s_0) V(x'(s_1), s_1) \tag{7.101}$$

where maximization over n_0 and the S elements of $x'(s_1)$ is subject to the time 0 implementability constraint

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + \beta \sum_{s_1 \in \mathcal{S}} \Pi(s_1|s_0) x'(s_1) \tag{7.102}$$

coming from restriction (7.97)

Associated with a value function $W(b_0, n_0)$ that solves Bellman equation are $S+1$ time 0 policy functions

$$\begin{aligned} n_0 &= f_0(b_0, s_0) \\ x_1(s_1) &= h_0(s_1; b_0, s_0). \end{aligned} \tag{7.103}$$

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 (7.100) for the time $t \geq 1$ continuation Ramsey planners.

The value function $V(x_t, s_t)$ of the time t continuation Ramsey planner equals $E_t \sum_{\tau=t}^{\infty} \beta^{\tau-t} u(c_t, l_t)$, where the consumption and leisure processes are evaluated along the original time 0 Ramsey plan.

First-order conditions

Attach a Lagrange multiplier $\Phi_1(x, s)$ to constraint (7.99) and a Lagrange multiplier Φ_0 to constraint (7.97).

Working backwards, the first-order conditions for the time $t \geq 1$ constrained maximization problem on the right side of the continuation Ramsey planner's Bellman equation (7.98) are

$$\beta \Pi(s'|s) V_x(x', s') - \beta \Pi(s'|s) \Phi_1 = 0 \tag{7.104}$$

for $x'(s')$ and

$$(1 + \Phi_1)(u_c - u_l) + \Phi_1 \left[n(u_{ll} - u_{lc}) + (n - g(s))(u_{cc} - u_{lc}) \right] = 0 \tag{7.105}$$

for n.

Given Φ_1, equation (7.105) is one equation to be solved for n as a function of s (or of $g(s)$).

Equation (7.104) implies $V_x(x', s') = \Phi_1$, while an envelope condition is $V_x(x, s) = \Phi_1$, so it follows that

$$V_x(x', s') = V_x(x, s) = \Phi_1(x, s). \tag{7.106}$$

For the time 0 problem on the right side of the Ramsey planner's Bellman equation (7.101), the first-order conditions are

$$V_x(x(s_1), s_1) = \Phi_0 \tag{7.107}$$

for $x(s_1), s_1 \in \mathcal{S}$, and

$$\begin{aligned} (1 + \Phi_0)(u_{c,0} - u_{n,0}) + \Phi_0 \big[n_0(u_{ll,0} - u_{lc,0}) + (n_0 - g(s_0))(u_{cc,0} - u_{cl,0}) \big] \\ - \Phi_0 (u_{cc,0} - u_{cl,0}) b_0 = 0 \end{aligned} \tag{7.108}$$

Notice similarities and differences between the first-order conditions for $t \geq 1$ and for $t = 0$.

An additional term is present in (7.108) except in the three special cases in which

- $b_0 = 0$, 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 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 (7.107) and (7.108) imply that $\Phi_0 = \Phi_1$ and that

$$V_x(x_t, s_t) = \Phi_0 \tag{7.109}$$

for all $t \geq 1$.

When V is concave in x, this implies *state-variable degeneracy* along a Ramsey plan in the sense that for $t \geq 1$, x_t will be a time-invariant function of s_t.

Given Φ_0, this function mapping s_t into x_t can be expressed as a vector $\vec{x}$ that solves equation (7.102) for n and c as functions of g that are associated with $\Phi = \Phi_0$.

Manifestations of time inconsistency

While the marginal utility adjusted level of government debt x_t is a key state variable for the continuation Ramsey planners at $t \geq 1$, it is not a state variable at time 0.

The time 0 Ramsey planner faces b_0, not $x_0 = u_{c,0} b_0$, as a state variable.

The discrepancy in state variables faced by the time 0 Ramsey planner and the time $t \geq 1$ continuation Ramsey planners captures the differing obligations and incentives faced by the time 0 Ramsey planner and the time $t \geq 1$ continuation Ramsey planners.

- The time 0 Ramsey planner is obligated to honor government debt b_0 measured in time 0 consumption goods
- The time 0 Ramsey planner can manipulate the *value* of government debt as measured by $u_{c,0} b_0$.
- In contrast, time $t \geq 1$ continuation Ramsey planners are obligated *not* to alter values of debt, as measured by $u_{c,t} b_t$, that they inherit from a preceding Ramsey planner or continuation Ramsey planner.

When government expenditures g_t are a time invariant function of a Markov state s_t, a Ramsey plan and associated Ramsey allocation feature marginal utilities of consumption $u_c(s_t)$ that, given Φ, for $t \geq 1$ depend only on s_t, but that for $t = 0$ depend on b_0 as well.

This means that $u_c(s_t)$ will be a time invariant function of s_t for $t \geq 1$, but except when $b_0 = 0$, a different function for $t = 0$.

This in turn means that prices of one period Arrow securities $p_t(s_{t+1}|s_t) = p(s_{t+1}|s_t)$ will be the *same* time invariant functions of (s_{t+1}, s_t) for $t \geq 1$, but a different function $p_0(s_1|s_0)$ for $t = 0$, except when $b_0 = 0$.

The differences between these time 0 and time $t \geq 1$ objects reflect the Ramsey planner's incentive to manipulate Arrow security prices and, through them, the value of initial government debt b_0.

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 $t < 3$ or $t > 3$ we assume that $g_t = g_l = 0.1$.
- At $t = 3$ a war occcurs with probability 0.5.
  - If there is war, $g_3 = g_h = 0.2$
  - If there is no war $g_3 = g_l = 0.1$

We define the components of the state vector as the following six (t, g) pairs: $(0, g_l), (1, g_l), (2, g_l), (3, g_l), (3, g_h), (t \geq 4, g_l)$.

We think of these 6 states as corresponding to $s = 1, 2, 3, 4, 5, 6$

The transition matrix is

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Government expenditures at each state are

$$g = \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \\ 0.1 \\ 0.2 \\ 0.1 \end{pmatrix}.$$

We assume that the representative agent has utility function

$$u(c, n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma}$$

and set $\sigma = 2$, $\gamma = 2.$, and the discount factor $\beta = 0.9$

Note: For convenience in terms of matching our computer code, we have expressed utility as a function of n rather than leisure l

---

We set initial government debt $b_0 = 1$

The following figure plots the Ramsey tax under both realizations of time $t = 3$ government expenditures

- black when $g_3 = .1$, and

- red when $g_3 = .2$

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

$$R_t = \frac{u_{c,t}}{\beta \mathbb{E}_t [u_{c,t+1}]}$$

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 figure illustrates how the government lowers the interest rate at time 0 by raising consumption.

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 then trades state-contingent Arrow securities to hedge against war at $t = 3$.

- It purchases a security that pays off when $g_3 = g_h$
- It sells a security that pays off when $g_3 = g_l$
- These purchases are designed in such a way that regardless of whether or not there is a war at $t = 3$, the government will begin period $t = 4$ with the *same* government debt

- The time $t = 4$ debt level can be serviced with revenues from the constant tax rate set at times $t \geq 1$

At times $t \geq 4$ the government rolls over its debt, knowing that the tax rate is set at 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 $g_t = g_l$ for all t

The figure below plots the Ramsey tax rates at time $t = 0$ and time $t \geq 1$ as functions of the initial government 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 $t \geq 0$

The first is $b_0 = 0$.

- Here the government can't use the $t = 0$ tax rate to alter the value of the initial debt

The second occurs when the government enters with sufficiently large assets that the Ramsey planner can achieve first best and sets $\tau_t = 0$ for all t

It is only for these two values of initial government debt that the Ramsey plan is time consistent.

Another way of saying this is that, except for these two values of initial government debt, a continuation of a Ramsey plan is not a Ramsey plan

To illustrate this, consider a Ramsey planner who starts with an initial government debt b_1 associated with one of the Ramsey plans computed above

Call τ_1^R the time $t = 0$ tax rate chosen by the Ramsey planner confronting this value for initial government debt government

The figure below shows both the tax rate at time 1 chosen by our original Ramsey planner and what the new Ramsey planner would choose for its time $t = 0$ tax rate

The tax rates in the figure are equal for only two values of initial government debt.

Tax Smoothing and non-CES preferences

The complete tax smoothing for $t \geq 1$ in the preceding example is a consequence of our having assumed CES preferences

To see what is driving this outcome, we begin by noting that the tax rate will be a time invariant function $\tau(\Phi, g)$ of the Lagrange multiplier on the implementability constraint and government expenditures

For CES preferences, we can exploit the relations $U_{cc}c = -\sigma U_c$ and $U_{nn}n = \gamma U_n$ to derive

$$\frac{(1 + (1 - \sigma)\Phi)U_c}{1 + (1 - \gamma)\Phi)U_n} = 1$$

from the first-order conditions.

This equation immediately implies that the tax rate is constant

For other preferences, the tax rate may not be constant.

For example, let the period utility function be

$$u(c, n) = \log(c) + 0.69 \log(1 - n)$$

and suppose that g_t follows a two state i.i.d. process with equal weights on g_l and g_h

The figure below plots a sample path of the Ramsey tax rate

We computed the tax rate by using both the sequential and recursive approaches described above

As should be expected, the recursive and sequential solutions produce almost identical allocations

Unlike outcomes with CES preferences, the tax rate is not perfectly smoothed

Instead the government raises the tax rate when g_t is high

Implementation

The following code implements the examples above

The module lucas_stokey.py solves and simulates a Ramsey plan using both the recursive and sequential techniques.

We implement both techniques as classes that take as an input an object called `Para`

`Para` sets numbers defining an economy, including a discount factor, a transition matrix, and a vector of government expenditures

It also contains the utility function U and its derivatives, which allows us easily to alter the period utility function

```
# -*- coding: utf-8 -*-
"""
Created on Wed Feb 18 15:43:37 2015

@author: dgevans
"""
import numpy as np
from scipy.optimize import root
from scipy.optimize import fmin_slsqp
from scipy.interpolate import UnivariateSpline
from quantecon import compute_fixed_point, MarkovChain


class Planners_Allocation_Sequential:
    '''
    Class returns planner's allocation as a function of the multiplier on the
    implementability constraint mu
    '''
    def __init__(self,Para):
        '''
        Initializes the class from the calibration Para
        '''
        self.beta = Para.beta
        self.Pi = Para.Pi
        self.mc = MarkovChain(self.Pi)
        self.G = Para.G
        self.S = len(Para.Pi) # number of states
        self.Theta = Para.Theta
        self.Para = Para
        #now find the first best allocation
        self.find_first_best()

    def find_first_best(self):
        '''
        Find the first best allocation
        '''
        Para = self.Para
        S,Theta,Uc,Un,G = self.S,self.Theta,Para.Uc,Para.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:]
        self.XiFB = Uc(self.cFB,self.nFB) #multiplier on the resource␣
↪constraint.
        self.zFB = np.hstack([self.cFB,self.nFB,self.XiFB])

    def time1_allocation(self,mu):
        '''
        Computes optimal allocation for time t\geq 1 for a given \mu
        '''
        Para = self.Para
        S,Theta,G,Uc,Ucc,Un,Unn = self.S,self.Theta,self.G,Para.Uc,Para.Ucc,
↪Para.Un,Para.Unn
        def FOC(z):
            c = z[:S]
            n = z[S:2*S]
            Xi = z[2*S:]
            return np.hstack([
            Uc(c,n) - mu*(Ucc(c,n)*c+Uc(c,n)) -Xi, #foc c
            Un(c,n) - mu*(Unn(c,n)*n+Un(c,n)) + Theta*Xi, #foc n
            Theta*n - c - G #resource constraint
            ])

        #find the root of the FOC
        res = root(FOC,self.zFB)
        if not res.success:
            raise Exception('Could not find LS allocation.')
        z = res.x
        c,n,Xi = z[:S],z[S:2*S],z[2*S:]

        #now compute x
        I  = Uc(c,n)*c +  Un(c,n)*n
        x = np.linalg.solve(np.eye(S) - self.beta*self.Pi, 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
        '''
        Para,Pi,Theta,G,beta = self.Para,self.Pi,self.Theta,self.G,self.beta
        Uc,Ucc,Un,Unn = Para.Uc,Para.Ucc,Para.Un,Para.Unn

        #first order conditions of planner's problem
        def FOC(z):
```

```
            mu,c,n,Xi = z
            xprime = self.time1_allocation(mu)[2]
            return np.hstack([
            Uc(c,n)*(c-B_) + Un(c,n)*n + beta*Pi[s_0].dot(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.Para.U(c,n)
        V = np.linalg.solve(np.eye(self.S) - self.beta*self.Pi, U )
        return c,n,x,V

    def Tau(self,c,n):
        '''
        Computes Tau given c,n
        '''
        Para = self.Para
        Uc,Un = Para.Uc(c,n),Para.Un(c,n)

        return 1+Un/(self.Theta * Uc)

    def simulate(self,B_,s_0,T,sHist=None):
        '''
        Simulates planners policies for T periods
        '''
        Para,Pi,beta = self.Para,self.Pi,self.beta
        Uc = Para.Uc


        if sHist == None:
            sHist = self.mc.simulate(T, s_0)

        cHist,nHist,Bhist,TauHist,muHist = np.zeros((5,T))
        RHist = np.zeros(T-1)
        #time0
        mu,cHist[0],nHist[0],_  = self.time0_allocation(B_,s_0)
        TauHist[0] = self.Tau(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)
            Tau = self.Tau(c,n)
            u_c = Uc(c,n)
            s = sHist[t]
            Eu_c = Pi[sHist[t-1]].dot(u_c)

            cHist[t],nHist[t],Bhist[t],TauHist[t] = c[s],n[s],x[s]/u_c[s],
↪Tau[s]

            RHist[t-1] = Uc(cHist[t-1],nHist[t-1])/(beta*Eu_c)
            muHist[t] = mu

        return cHist,nHist,Bhist,TauHist,sHist,muHist,RHist




class Planners_Allocation_Bellman:
    '''
    Compute the planner's allocation by solving Bellman
    equation.
    '''
    def __init__(self,Para,mugrid):
        '''
        Initializes the class from the calibration Para
        '''
        self.beta = Para.beta
        self.Pi = Para.Pi
        self.mc = MarkovChain(self.Pi)
        self.G = Para.G
        self.S = len(Para.Pi) # number of states
        self.Theta = Para.Theta
        self.Para = Para
        self.mugrid = mugrid

        #now 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 Para and initial␣
↪grid mugrid0
        '''
        Para,mugrid0 = self.Para,self.mugrid
        S = len(Para.Pi)

        #First get initial fit
        PP = Planners_Allocation_Sequential(Para)
        c,n,x,V = map(np.vstack, zip(*map(lambda mu: PP.time1_value(mu),
↪mugrid0)) )
```

```
        Vf,cf,nf,xprimef = {},{},{},{}
        for s in range(2):
            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(mugrid0))
        self.xgrid = xgrid

        #Now iterate on bellman equation
        T = BellmanEquation(Para,xgrid,policies)
        diff = 1.
        while diff > 1e-5:
            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() )

            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 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 Tau(self,c,n):
        '''
        Computes Tau given c,n
        '''
        Para = self.Para
        Uc,Un = Para.Uc(c,n),Para.Un(c,n)

        return 1+Un/(self.Theta * 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
        '''
        Para,Pi = self.Para,self.Pi
        Uc = Para.Uc
        cf,nf,xprimef = self.policies

        if sHist == None:
            sHist = self.mc.simulate(T, s_0)

        cHist,nHist,Bhist,TauHist,muHist = np.zeros((5,T))
        RHist = np.zeros(T-1)
        #time0
        cHist[0],nHist[0],xprime  = self.time0_allocation(B_,s_0)
        TauHist[0] = self.Tau(cHist[0],nHist[0])[s_0]
        Bhist[0] = B_
        muHist[0] = 0.

        #time 1 onward
        for t in range(1,T):
            s,x = sHist[t],xprime[sHist[t]]
            c,n,xprime = np.empty(self.S),nf[s](x),np.empty(self.S)
            for shat in range(self.S):
                c[shat] = cf[shat](x)
            for sprime in range(self.S):
                xprime[sprime] = xprimef[s,sprime](x)

            Tau = self.Tau(c,n)[s]
            u_c = Uc(c,n)
```

```
            Eu_c = Pi[sHist[t-1]].dot(u_c)
            muHist[t] = self.Vf[s](x,1)

            RHist[t-1] = Uc(cHist[t-1],nHist[t-1])/(self.beta*Eu_c)

            cHist[t],nHist[t],Bhist[t],TauHist[t] = c[s],n,x/u_c[s],Tau

        return cHist,nHist,Bhist,TauHist,sHist,muHist,RHist

class BellmanEquation(object):
    '''
    Bellman equation for the continuation of the Lucas-Stokey Problem
    '''
    def __init__(self,Para,xgrid,policies0):
        '''
        Initializes the class from the calibration Para
        '''
        self.beta = Para.beta
        self.Pi = Para.Pi
        self.G = Para.G
        self.S = len(Para.Pi) # number of states
        self.Theta = Para.Theta
        self.Para = Para

        self.xbar = [min(xgrid),max(xgrid)]
        self.time_0 = False

        self.z0 = {}
        cf,nf,xprimef = policies0
        for s in range(self.S):
            for x in xgrid:
                xprime0 = np.empty(self.S)
                for sprime in range(self.S):
                    xprime0[sprime] = xprimef[s,sprime](x)
                self.z0[x,s] = np.hstack([cf[s](x),nf[s](x),xprime0])

        self.find_first_best()

    def find_first_best(self):
        '''
        Find the first best allocation
        '''
        Para = self.Para
        S,Theta,Uc,Un,G = self.S,self.Theta,Para.Uc,Para.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.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:
            PF = lambda x,s: self.get_policies_time1(x,s,Vf)
        else:
            PF = lambda B_,s0: self.get_policies_time0(B_,s0,Vf)
        return PF

    def get_policies_time1(self,x,s,Vf):
        '''
        Finds the optimal policies
        '''
        Para,beta,Theta,G,S,Pi = self.Para,self.beta,self.Theta,self.G,self.S,
↪self.Pi
        U,Uc,Un = Para.U,Para.Uc,Para.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].dot(Vprime))

        def cons(z):
            c,n,xprime = z[0],z[1],z[2:]
            return np.hstack([
            x - Uc(c,n)*c-Un(c,n)*n - beta*Pi[s].dot(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)
```

```
        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
        '''
        Para,beta,Theta,G,S,Pi = self.Para,self.beta,self.Theta,self.G,self.S,
↪self.Pi
        U,Uc,Un = Para.U,Para.Uc,Para.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].dot(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].dot(xprime),
            (Theta*n - c - G)[s0]
            ])


        out,fx,_,imode,smode = fmin_slsqp(objf,self.zFB[s0],f_eqcons=cons,
                                bounds=[(0.,100),(0.,100)]+[self.xbar]*S,full_
↪output=True,iprint=0)

        if imode >0:
            raise Exception(smode)

        return np.hstack([-fx,out])
```

Numbers defining economies are in CES.py and BGP.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 10 08:55:03 2015

@author: dgevans
"""

import numpy as np

class baseline_CES(object):
    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

    #derivatives of utiltiy 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)

    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.)

#Model 1
M1_CES = baseline_CES()

#Model 2

M2_CES = baseline_CES()
M2_CES.G = np.array([0.15])
M2_CES.Pi = np.ones((1,1))
M2_CES.Theta = np.ones(1)

#Model 3 with time varying

M_time_example_CES = baseline_CES()

M_time_example_CES.Pi = 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.]])

M_time_example_CES.G = np.array([0.1, 0.1, 0.1, 0.2, 0.1, 0.1])
M_time_example_CES.Theta = np.ones(6) # Theta can in principle be random
```

```
# -*- coding: utf-8 -*-
"""
Created on Wed Feb 18 15:32:07 2015

@author: dgevans
"""
import numpy as np

class baseline_BGP(object):
    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

    #derivatives of utiltiy function
    def U(self,c,n):
        return np.log(c) + self.psi*np.log(1-n)

    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


#Model 1
M1_BGP = baseline_BGP()

#Model 2

M2_BGP = baseline_BGP()
M2_BGP.G = np.array([0.15])
M2_BGP.Pi = np.ones((1,1))
M2_BGP.Theta = np.ones(1)

#Model 3 with time varying
```

```
M_time_example_BGP = baseline_BGP()

M_time_example_BGP.Pi = 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.]])

M_time_example_BGP.G = np.array([0.1, 0.1, 0.1, 0.2, 0.1, 0.1])
M_time_example_BGP.Theta = np.ones(6) # Theta can in principle be random
```

The file main_LS.py solves and plots the examples.

```
# -*- coding: utf-8 -*-
"""
Created on Fri Feb 20 14:07:56 2015

@author: dgevans
"""
import matplotlib.pyplot as plt
import numpy as np


'''
Time Varying Example
'''

PP_seq_time = Planners_Allocation_Sequential(M_time_example_CES) #solve␣
↪sequential 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 = PP_seq_time.simulate(1.,0,7,sHist_h)
sim_seq_l = PP_seq_time.simulate(1.,0,7,sHist_l)

plt.figure(figsize=[14,10])
plt.subplot(3,2,1)
plt.title('Consumption')
plt.plot(sim_seq_l[0],'-ok')
plt.plot(sim_seq_h[0],'-or')
plt.subplot(3,2,2)
plt.title('Labor Supply')
plt.plot(sim_seq_l[1],'-ok')
plt.plot(sim_seq_h[1],'-or')
plt.subplot(3,2,3)
plt.title('Government Debt')
plt.plot(sim_seq_l[2],'-ok')
plt.plot(sim_seq_h[2],'-or')
plt.subplot(3,2,4)
plt.title('Taxe Rate')
```

```
plt.plot(sim_seq_l[3],'-ok')
plt.plot(sim_seq_h[3],'-or')
plt.subplot(3,2,5)
plt.title('Government Spending')
plt.plot(M_time_example_CES.G[sHist_l],'-ok')
plt.plot(M_time_example_CES.G[sHist_h],'-or')
plt.subplot(3,2,6)
plt.title('Output')
plt.plot(M_time_example_CES.Theta[sHist_l]*sim_seq_l[1],'-ok')
plt.plot(M_time_example_CES.Theta[sHist_h]*sim_seq_h[1],'-or')

plt.tight_layout()
plt.savefig('TaxSequence_time_varying.png')

plt.figure(figsize=[8,5])
plt.title('Gross Interest Rate')
plt.plot(sim_seq_l[-1],'-ok')
plt.plot(sim_seq_h[-1],'-or')
plt.tight_layout()
plt.savefig('InterestRate_time_varying.png')
plt.show()

'''
Time 0 example
'''
PP_seq_time0 = Planners_Allocation_Sequential(M2_CES) #solve sequential␣
↪problem

B_vec = np.linspace(-1.5,1.,100)
taxpolicy = np.vstack([PP_seq_time0.simulate(B_,0,2)[3] for B_ in B_vec])
interest_rate = np.vstack([PP_seq_time0.simulate(B_,0,3)[-1] for B_ in B_vec])

plt.figure(figsize=[14,6])
plt.subplot(211)
plt.plot(B_vec,taxpolicy[:,0],linewidth=2.)
plt.plot(B_vec,taxpolicy[:,1],linewidth=2.)

plt.title('Tax Rate')
plt.legend((r'Time $t=0$', 'Time $t\geq1$'),loc=2,shadow=True)
plt.subplot(212)
plt.title('Gross Interest Rate')
plt.plot(B_vec,interest_rate[:,0],linewidth=2.)
plt.plot(B_vec,interest_rate[:,1],linewidth=2.)
plt.xlabel('Initial Government Debt')
plt.tight_layout()

plt.savefig('Time0_taxpolicy.png')
plt.show()




#compute the debt entered with at time 1
```

```
B1_vec = np.hstack([PP_seq_time0.simulate(B_,0,2)[2][1] for B_ in B_vec])
#now compute the optimal policy if the government could reset
tau1_reset = np.hstack([PP_seq_time0.simulate(B1,0,1)[3] for B1 in B1_vec])

plt.figure(figsize=[10,6])
plt.plot(B_vec,taxpolicy[:,1],linewidth=2.)
plt.plot(B_vec,tau1_reset,linewidth=2.)
plt.xlabel('Initial Government Debt')
plt.title('Tax Rate')
plt.legend((r'$\tau_1$', r'$\tau_1^R$'),loc=2,shadow=True)
plt.tight_layout()

plt.savefig('Time0_inconsistent.png')
plt.show()


'''
BGP Example
'''
#initialize mugrid for value function iteration
muvec = np.linspace(-0.6,0.0,200)


PP_seq = Planners_Allocation_Sequential(M1_BGP) #solve sequential problem
PP_bel = Planners_Allocation_Bellman(M1_BGP,muvec) #solve recursive problem

T = 20
#sHist = utilities.simulate_markov(M1_BGP.Pi,0,T)
sHist = np.array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0],
↪dtype=int)

#simulate
sim_seq = PP_seq.simulate(0.5,0,T,sHist)
sim_bel = PP_bel.simulate(0.5,0,T,sHist)

#plot policies
plt.figure(figsize=[14,10])
plt.subplot(3,2,1)
plt.title('Consumption')
plt.plot(sim_seq[0],'-ok')
plt.plot(sim_bel[0],'-xk')
plt.legend(('Sequential','Recursive'),loc='best')
plt.subplot(3,2,2)
plt.title('Labor Supply')
plt.plot(sim_seq[1],'-ok')
plt.plot(sim_bel[1],'-xk')
plt.subplot(3,2,3)
plt.title('Government Debt')
plt.plot(sim_seq[2],'-ok')
plt.plot(sim_bel[2],'-xk')
plt.subplot(3,2,4)
plt.title('Tax Rate')
plt.plot(sim_seq[3],'-ok')
```

```
plt.plot(sim_bel[3],'-xk')
plt.subplot(3,2,5)
plt.title('Government Spending')
plt.plot(M1_BGP.G[sHist],'-ok')
plt.plot(M1_BGP.G[sHist],'-xk')
plt.plot(M1_BGP.G[sHist],'-^k')
plt.subplot(3,2,6)
plt.title('Output')
plt.plot(M1_BGP.Theta[sHist]*sim_seq[1],'-ok')
plt.plot(M1_BGP.Theta[sHist]*sim_bel[1],'-xk')

plt.tight_layout()
plt.savefig('TaxSequence_LS.png')

plt.figure(figsize=[8,5])
plt.title('Gross Interest Rate')
plt.plot(sim_seq[-1],'-ok')
plt.plot(sim_bel[-1],'-xk')
plt.legend(('Sequential','Recursive'),loc='best')
plt.tight_layout()
plt.show()
```

Further Comments

A *related lecture* describes an extension of the Lucas-Stokey model by Aiyagari, Marcet, Sargent, and Seppälä (2002) *[AMSS02]*

In their (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

Optimal Taxation in an LQ Economy

Contents

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

– *Examples*
– *Exercises*
– *Solutions*

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 government's tax rate and borrowing plans as given

Maximum attainable utility for the household depends on the government's tax and borrowing plans

The *Ramsey problem* *[Ram27]* is to choose tax and borrowing plans that maximize the household's welfare, taking the household's optimizing behavior as given

There is a large number of competitive equilibria indexed by different government fiscal policies

The Ramsey planner chooses the best competitive equilibrium

We want to study the dynamics of tax rates, tax revenues, government debt under a Ramsey plan

Because the Lucas and Stokey model features state-contingent government debt, the government debt dynamics differ substantially from those in a model of Robert Barro *[Bar79]*

The treatment given here closely follows this manuscript, prepared by Thomas J. Sargent and Francois R. Velde

We cover only the key features of the problem in this lecture, leaving you to refer to that source for additional results and intuition

Model Features

- Linear quadratic (LQ) model
- Representative household
- Stochastic dynamic programming over an infinite horizon
- Distortionary taxation

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

$$-\mathbb{E}\frac{1}{2}\sum_{t=0}^{\infty}\beta^t\left[(c_t-b_t)^2+\ell_t^2\right] \tag{7.110}$$

subject to the budget constraint

$$\mathbb{E}\sum_{t=0}^{\infty}\beta^t p_t^0\left[d_t+(1-\tau_t)\ell_t+s_t-c_t\right]=0 \tag{7.111}$$

Here

- β is a discount factor in $(0,1)$
- p_t^0 is a scaled Arrow-Debreu price at time 0 of history contingent goods at time $t+j$
- b_t is a stochastic preference parameter
- d_t is an endowment process
- τ_t is a flat tax rate on labor income
- s_t is a promised time-t coupon payment on debt issued by the government

The scaled Arrow-Debreu price p_t^0 is related to the unscaled Arrow-Debreu price as follows.

If we let $\pi_t^0(x^t)$ denote the probability (density) of a history $x^t=[x_t,x_{t-1},\ldots,x_0]$ 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

$$\frac{\beta^t p_t^0}{\pi_t^0(x^t)}$$

Thus, our scaled Arrow-Debreu price is the ordinary Arrow-Debreu price multiplied by the discount factor β^t and divided by an appropriate probability.

The budget constraint (7.111) 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 $x_{t+1} = Ax_t + Cw_{t+1}$ where $\{w_t\}$ is independent zero mean Gaussian with identify covariance matrix

Feasibility

The period-by-period feasibility restriction for this economy is

$$c_t + g_t = d_t + \ell_t \tag{7.112}$$

A labor-consumption process $\{\ell_t, c_t\}$ is called *feasible* if (7.112) holds for all t

Government budget constraint

Where p_t^0 is again a scaled Arrow-Debreu price, the time zero government budget constraint is

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t p_t^0 (s_t + g_t - \tau_t \ell_t) = 0 \tag{7.113}$$

Equilibrium

An *equilibrium* is a feasible allocation $\{\ell_t, c_t\}$, a sequence of prices $\{p_t^0\}$, and a tax system $\{\tau_t\}$ such that

1. The allocation $\{\ell_t, c_t\}$ is optimal for the household given $\{p_t^0\}$ and $\{\tau_t\}$
2. The government's budget constraint (7.113) is satisfied

The *Ramsey problem* is to choose the equilibrium $\{\ell_t, c_t, \tau_t, p_t^0\}$ that maximizes the household's welfare

If $\{\ell_t, c_t, \tau_t, p_t^0\}$ solves the Ramsey problem, then $\{\tau_t\}$ is called the *Ramsey plan*

The solution procedure we adopt is

1. Use the first-order conditions from the household problem to pin down prices and allocations given $\{\tau_t\}$
2. Use these expressions to rewrite the government budget constraint (7.113) in terms of exogenous variables and allocations
3. Maximize the household's objective function (7.110) subject to the constraint constructed in step 2 and the feasibility constraint (7.112)

The solution to this maximization problem pins down all quantities of interest

Solution

Step one is to obtain the first-conditions for the household's problem, taking taxes and prices as given

Letting μ be the Lagrange multiplier on (7.111), the first-order conditions are $p_t^0 = (c_t - b_t)/\mu$ and $\ell_t = (c_t - b_t)(1 - \tau_t)$

Rearranging and normalizing at $\mu = b_0 - c_0$, we can write these conditions as

$$p_t^0 = \frac{b_t - c_t}{b_0 - c_0} \quad \text{and} \quad \tau_t = 1 - \frac{\ell_t}{b_t - c_t} \tag{7.114}$$

Substituting (7.114) into the government's budget constraint (7.113) yields

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t \left[(b_t - c_t)(s_t + g_t - \ell_t) + \ell_t^2 \right] = 0 \tag{7.115}$$

The Ramsey problem now amounts to maximizing (7.110) subject to (7.115) and (7.112)

The associated Lagrangian is

$$\mathscr{L} = \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left\{ -\frac{1}{2} \left[(c_t - b_t)^2 + \ell_t^2 \right] + \lambda \left[(b_t - c_t)(\ell_t - s_t - g_t) - \ell_t^2 \right] + \mu_t [d_t + \ell_t - c_t - g_t] \right\} \tag{7.116}$$

The first order conditions associated with c_t and ℓ_t are

$$-(c_t - b_t) + \lambda[-\ell_t + (g_t + s_t)] = \mu_t$$

and

$$\ell_t - \lambda[(b_t - c_t) - 2\ell_t] = \mu_t$$

Combining these last two equalities with (7.112) and working through the algebra, one can show that

$$\ell_t = \bar{\ell}_t - \nu m_t \quad \text{and} \quad c_t = \bar{c}_t - \nu m_t \tag{7.117}$$

where

- $\nu := \lambda/(1 + 2\lambda)$
- $\bar{\ell}_t := (b_t - d_t + g_t)/2$
- $\bar{c}_t := (b_t + d_t - g_t)/2$
- $m_t := (b_t - d_t - s_t)/2$

Apart from ν, all of these quantities are expressed in terms of exogenous variables

To solve for ν, we can use the government's budget constraint again

The term inside the brackets in (7.115) is $(b_t - c_t)(s_t + g_t) - (b_t - c_t)\ell_t + \ell_t^2$

Using (7.117), 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 (7.115), we get

$$\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t (b_t - \bar{c}_t)(g_t + s_t)\right\} + (\nu^2 - \nu)\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t 2m_t^2\right\} = 0 \tag{7.118}$$

Although it might not be clear yet, we are nearly there because:

- The two expectations terms in (7.118) can be solved for in terms of model primitives
- This in turn allows us to solve for the Lagrange multiplier ν
- With ν in hand, we can go back and solve for the allocations via (7.117)
- Once we have the allocations, prices and the tax system can be derived from (7.114)

Computing the Quadratic Term

Let's consider how to obtain the term ν in (7.118)

If we can compute the two expected geometric sums

$$b_0 := \mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t (b_t - \bar{c}_t)(g_t + s_t)\right\} \quad \text{and} \quad a_0 := \mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t 2m_t^2\right\} \tag{7.119}$$

then the problem reduces to solving

$$b_0 + a_0(\nu^2 - \nu) = 0$$

for ν

Provided that $4b_0 < a_0$, there is a unique solution $\nu \in (0, 1/2)$, and a unique corresponding $\lambda > 0$

Let's work out how to compute mathematical expectations in (7.119)

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 (7.119), the random variable $2m_t^2$ can be written as

$$\frac{1}{2} x_t'(S_b - S_d - S_s)'(S_b - S_d - S_s)x_t$$

It follows that both objects of interest are special cases of the expression

$$q(x_0) = \mathbb{E} \sum_{t=0}^{\infty} \beta^t x_t' H x_t \tag{7.120}$$

where H is a matrix conformable to x_t and x_t' is the transpose of column vector x_t

Suppose first that $\{x_t\}$ is the Gaussian VAR described *above*

In this case, the formula for computing $q(x_0)$ is known to be $q(x_0) = x_0' Q x_0 + v$, where

- Q is the solution to $Q = H + \beta A' Q A$, and
- $v = \text{trace}\,(C'QC)\beta/(1-\beta)$

The first equation is known as a discrete Lyapunov equation, and can be solved using this function

Finite state Markov case

Next suppose that $\{x_t\}$ is the discrete Markov process described *above*

Suppose further that each x_t takes values in the state space $\{x^1, \ldots, x^N\} \subset \mathbb{R}^k$

Let $h \colon \mathbb{R}^k \to \mathbb{R}$ be a given function, and suppose that we wish to evaluate

$$q(x_0) = \mathbb{E} \sum_{t=0}^{\infty} \beta^t h(x_t) \quad \text{given} \quad x_0 = x^j$$

For example, in the discussion above, $h(x_t) = x_t' H x_t$

It is legitimate to pass the expectation through the sum, leading to

$$q(x_0) = \sum_{t=0}^{\infty} \beta^t (P^t h)[j] \tag{7.121}$$

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 (7.121) 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

$$p_{t+j}^t = \frac{b_{t+j} - c_{t+j}}{b_t - c_t}$$

as the scaled Arrow-Debreu time t price of a history contingent claim on one unit of consumption at time $t + j$

These are prices that would prevail at time t if 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

$$B_t := \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j p_{t+j}^t (\tau_{t+j} \ell_{t+j} - g_{t+j}) \tag{7.122}$$

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 verity that (7.122) implies that

$$B_t = (\tau_t \ell_t - g_t) + E_t \sum_{j=1}^{\infty} p_{t+j}^t (\tau_{t+j} \ell_{t+j} - g_{t+j})$$

and

$$B_t = (\tau_t \ell_t - g_t) + \beta E_t p_{t+1}^t B_{t+1} \tag{7.123}$$

Define

$$R_t^{-1} := \mathbb{E}_t \beta^j p_{t+1}^t \tag{7.124}$$

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 π_t

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

The term π_{t+1} is the difference between two quantities:

- B_{t+1}, the value of government debt at the start of period $t+1$
- $R_t[B_t + g_t - \tau_t]$, which is what the government would have owed at the beginning of period $t+1$ if it had simply borrowed at the one-period risk-free rate rather than selling state-contingent securities.

Thus, π_{t+1} is the excess payout on the actual portfolio of state contingent government debt relative to an alternative portfolio sufficient to finance $B_t + g_t - \tau_t \ell_t$ and consisting entirely of risk-free one-period bonds.

Use expressions (7.123) and (7.124) to obtain

$$\pi_{t+1} = B_{t+1} - \frac{1}{\beta E_t p_{t+1}^t} \left[\beta E_t p_{t+1}^t B_{t+1}\right]$$

or

$$\pi_{t+1} = B_{t+1} - \tilde{E}_t B_{t+1} \tag{7.125}$$

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 (7.125) 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-smoothng 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.

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

The file is `lqramsey.py`, and can be downloaded here

Description and clarifications are given below

```
"""
Filename: lqramsey.py
Authors: Thomas Sargent, Doc-Jin Jang, Jeong-hun Choi, John Stachurski

This module provides code to compute Ramsey equilibria in a LQ economy with
distortionary taxation.  The program computes allocations (consumption,
leisure), tax rates, revenues, the net present value of the debt and other
related quantities.

Functions for plotting the results are also provided below.

See the lecture at http://quant-econ.net/py/lqramsey.html for a description of
the model.

"""

import sys
import numpy as np
from numpy import sqrt, eye, dot, 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
        'tau',          # Tax rate
        'rvn',          # Revenue
        'B',            # Govt debt
        'R',            # Risk free gross return
        'pi',           # One-period risk-free interest rate
```

```
        'Pi',           # Cumulative rate of return, adjusted
        'xi'))          # Adjustment factor for Pi


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
         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 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
         tau          - Tax rate
         rvn          - Revenue
         B            - Govt debt
         R            - Risk free gross return
         pi           - One-period risk-free interest rate
         Pi           - Cumulative rate of return, adjusted
         xi           - Adjustment factor for Pi

        The corresponding values are flat numpy ndarrays.

    """

    # == Simplify names == #
    beta, Sg, Sd, Sb, Ss = econ.beta, econ.Sg, econ.Sd, econ.Sb, econ.Ss

    if econ.discrete:
        P, x_vals = econ.proc
    else:
        A, C = econ.proc

    # == Simulate the exogenous process x == #
```

```
    if econ.discrete:
        state = mc_sample_path(P, init=0, sample_size=T)
        x = x_vals[:, state]
    else:
        # == Generate an initial condition x0 satisfying x0 = A x0 == #
        nx, nx = A.shape
        x0 = nullspace((eye(nx) - A))
        x0 = -x0 if (x0[nx-1] < 0) else x0
        x0 = x0 / x0[nx-1]

        # == Generate a time series x of length T starting from x0 == #
        nx, nw = C.shape
        x = zeros((nx, T))
        w = randn(nw, T)
        x[:, 0] = x0.T
        for t in range(1, T):
            x[:, t] = dot(A, x[:, t-1]) + dot(C, w[:, t])

    # == Compute exogenous variable sequences == #
    g, d, b, s = (dot(S, x).flatten() for S in (Sg, Sd, Sb, Ss))

    # == Solve for Lagrange multiplier in the govt budget constraint == #
    # In fact we solve for nu = lambda / (1 + 2*lambda).  Here nu is the
    # solution to a quadratic equation a(nu**2 - nu) + 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(np.identity(ns) - beta * P)
        a0 = 0.5 * dot(F, dot(Sm, x_vals).T**2)[0]
        H = dot(Sb - Sd + Sg, x_vals) * dot(Sg - Ss, x_vals)
        b0 = 0.5 * dot(F, H.T)[0]
        a0, b0 = float(a0), float(b0)
    else:
        H = dot(Sm.T, Sm)
        a0 = 0.5 * var_quadratic_sum(A, C, H, beta, x0)
        H = dot((Sb - Sd + Sg).T, (Sg + Ss))
        b0 = 0.5 * var_quadratic_sum(A, C, H, beta, x0)

    # == Test that nu 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:
        nu = 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 nu * (0.5 - nu) < 0:
        print("Negative multiplier on the government budget constraint.")
        print(warning_msg.format('low', 'Democratic'))
        sys.exit(0)

    # == Solve for the allocation given nu and x == #
    Sc = 0.5 * (Sb + Sd - Sg - nu * Sm)
    Sl = 0.5 * (Sb - Sd + Sg - nu * Sm)
    c = dot(Sc, x).flatten()
    l = dot(Sl, x).flatten()
    p = dot(Sb - Sc, x).flatten()  # Price without normalization
    tau = 1 - l / (b - c)
    rvn = l * tau

    # == Compute remaining variables == #
    if econ.discrete:
        H = dot(Sb - Sc, x_vals) * dot(Sl - Sg, x_vals) - dot(Sl, x_vals)**2
        temp = dot(F, H.T).flatten()
        B = temp[state] / p
        H = dot(P[state, :], dot(Sb - Sc, x_vals).T).flatten()
        R = p / (beta * H)
        temp = dot(P[state, :], dot(Sb - Sc, x_vals).T).flatten()
        xi = p[1:] / temp[:T-1]
    else:
        H = dot(Sl.T, Sl) - dot((Sb - Sc).T, Sl - Sg)
        L = np.empty(T)
        for t in range(T):
            L[t] = var_quadratic_sum(A, C, H, beta, x[:, t])
        B = L / p
        Rinv = (beta * dot(dot(Sb - Sc, A), x)).flatten() / p
        R = 1 / Rinv
        AF1 = dot(Sb - Sc, x[:, 1:])
        AF2 = dot(dot(Sb - Sc, A), x[:, :T-1])
        xi = AF1 / AF2
        xi = xi.flatten()

    pi = B[1:] - R[:T-1] * B[:T-1] - rvn[:T-1] + g[:T-1]
    Pi = cumsum(pi * xi)

    # == Prepare return values == #
    path = Path(g=g,
            d=d,
            b=b,
            s=s,
            c=c,
            l=l,
            p=p,
            tau=tau,
            rvn=rvn,
            B=B,
            R=R,
            pi=pi,
```

```
            Pi=Pi,
            xi=xi)

    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(r'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=r'$g_t$', **p_args)
    ax.plot(path.c, label=r'$c_t$', **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=r'$g_t$', **p_args)
    ax.plot(list(range(1, T)), path.B[1:T], label=r'$B_{t+1}$', **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'$R_t - 1$', **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=r'$g_t$', **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.xi, label=r'$\xi_t$', **p_args)
    ax.grid()
    ax.set_xlabel(r'Time')
    ax.legend(ncol=1, **legend_args)

    # == Plot adjusted cumulative return == #
    ax = axes[1]
    ax.plot(list(range(2, T+1)), path.Pi, label=r'$\Pi_t$', **p_args)
    ax.grid()
    ax.set_xlabel(r'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 (7.120) when the exogenous process $\{x_t\}$ is of the VAR type described *above*

Below the definition of the function, you will see definitions of two `namedtuple` objects, `Economy` and `Path`

The first is used to collect all the parameters and primitives of a given LQ economy, while the second collects output of the computations

In Python, a `namedtuple` is a popular data type from the `collections` module of the standard library that replicates the functionality of a tuple, but also allows you to assign a name to each tuple element

These elements can then be references via dotted attribute notation — see for example the use of `path` in the functions `gen_fig_1()` and `gen_fig_2()`

The benefits of using `namedtuples`:

- Keeps content organized by meaning
- Helps reduce the number of global variables

Other than that, our code is long but relatively straightforward

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, from file lqramsey_ar1.py

```
"""
Filename: lqramsey_ar1.py
Authors: Thomas Sargent, Doc-Jin Jang, Jeong-hun Choi, John Stachurski

Example 1: Govt spending is AR(1) and state is (g, 1).

"""

import numpy as np
from numpy import array
#import lqramsey

# == Parameters == #
beta = 1 / 1.05
rho, mg = .7, .35
A = np.identity(2)
A[0, :] = rho, mg * (1-rho)
C = np.zeros((2, 1))
C[0, 0] = np.sqrt(1 - rho**2) * mg / 10
Sg = array((1, 0)).reshape(1, 2)
Sd = array((0, 0)).reshape(1, 2)
Sb = array((0, 2.135)).reshape(1, 2)
Ss = array((0, 0)).reshape(1, 2)
```

```
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)
```

Running the program produces the figure

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

After running the code above, if you then execute `lqramsey.gen_fig_2(path)` from your IPython shell you will produce the figure

See the original manuscript for comments and interpretation

The Discrete Case

Our second example adopts a discrete Markov specification for the exogenous process

Here's the code, from file lqramsey_discrete.py

```
"""
Filename: lqramsey_discrete.py
Authors: Thomas Sargent, Doc-Jin Jang, Jeong-hun Choi, John Stachurski

LQ Ramsey model with discrete exogenous process.

"""
```

```
from numpy import array
#import lqramsey

# == Parameters == #
beta = 1 / 1.05
P = array([[0.8, 0.2, 0.0],
           [0.0, 0.5, 0.5],
           [0.0, 0.0, 1.0]])
# == Possible states of the world == #
# Each column is a state of the world. The rows are [g d b s 1]
x_vals = 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 = array((1, 0, 0, 0, 0)).reshape(1, 5)
Sd = array((0, 1, 0, 0, 0)).reshape(1, 5)
Sb = array((0, 0, 1, 0, 0)).reshape(1, 5)
Ss = array((0, 0, 0, 1, 0)).reshape(1, 5)

economy = Economy(beta=beta,
                          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_1(path)` generates the figure

while `gen_fig_2(path)` generates

See the original manuscript for comments and interpretation

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

Solutions

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

Exercise 1

```
from numpy import array

# == Parameters == #
beta = 1 / 1.05
rho, mg = .95, .35
A = 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 = array((1, 0, 0, 0, 0)).reshape(1, 5)
Sd = array((0, 0, 0, 0, 0)).reshape(1, 5)
Sb = array((0, 0, 0, 0, 2.135)).reshape(1, 5)  # Chosen st. (Sc + Sg) * x0 = 1
Ss = 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)
```

Optimal Taxation without State-Contingent Debt

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 the assumptions and equilibrium concepts
- solve the model
- implement the model numerically and
- conduct some policy experiments
- compare outcomes with those in a complete-markets model

We begin with an introduction to the model.

A 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 the things that are identical

For $t \geq 0$, the 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.

A representative household is endowed with one unit of time each period that can be divided between leisure ℓ_t and labor n_t:

$$n_t(s^t) + \ell_t(s^t) = 1. \tag{7.126}$$

Output equals $n_t(s^t)$ and can be divided between consumption $c_t(s^t)$ and $g_t(s_t)$

$$c_t(s^t) + g_t(s_t) = n_t(s^t). \tag{7.127}$$

A representative household's preferences over $\{c_t(s^t), \ell_t(s^t)\}_{t=0}^{\infty}$ are ordered by

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)], \tag{7.128}$$

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 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

Lucas and Stokey assumed that there are complete markets in one-period Arrow securities.

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

Risk-free 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[1]

That $b_{t+1}(s^t)$ is the same for all realizations of s_{t+1} captures its *risk-free* quality

The market value at time t of government debt equals $b_{t+1}(s^t)$ divided by $R_t(s^t)$.

The government's budget constraint in period t at history s^t is

$$\begin{aligned} b_t(s^{t-1}) &= \tau_t^n(s^t)n_t(s^t) - g_t(s_t) - T_t(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)} \\ &\equiv z(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)}, \end{aligned} \tag{7.129}$$

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

$$\frac{1}{R_t(s^t)} = \sum_{s_{t+1}} p_t(s_{t+1}|s^t) = \sum_{s^{t+1}|s^t} \beta\pi_{t+1}(s^{t+1}|s^t)\frac{u_c(s^{t+1})}{u_c(s^t)}.$$

Substituting this expression into the government's budget constraint (7.129) yields:

$$b_t(s^{t-1}) = z(s^t) + \beta \sum_{s^{t+1}|s^t} \pi_{t+1}(s^{t+1}|s^t)\frac{u_c(s^{t+1})}{u_c(s^t)}\ b_{t+1}(s^t). \tag{7.130}$$

We will now replace this constant $b_{t+1}(s^t)$ by another expression of the same magnitude.

In fact, we have as many candidate expressions for that magnitude as there are possible states s_{t+1}

That is,

- For each state s_{t+1} there is a government budget constraint that is the analogue to expression (7.130) where the time index is moved one period forward.
- All of those budget constraints have a right side that equals $b_{t+1}(s^t)$.

Instead of picking one of these candidate expressions to replace all occurrences of $b_{t+1}(s^t)$ in equation (7.130), we replace $b_{t+1}(s^t)$ when the summation index in equation is s_{t+1} by the right side of next period's budget constraint that is associated with that particular realization s_{t+1}.

These substitutions give rise to the following expression:

$$b_t(s^{t-1}) = z(s^t) + \sum_{s^{t+1}|s^t} \beta\pi_{t+1}(s^{t+1}|s^t)\frac{u_c(s^{t+1})}{u_c(s^t)}\ \left[z(s^{t+1}) + \frac{b_{t+2}(s^{t+1})}{R_{t+1}(s^{t+1})}\right].$$

[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? 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.

After similar repeated substitutions for all future occurrences of government indebtedness, and by invoking the natural debt limit, we arrive at:

$$\begin{aligned} b_t(s^{t-1}) &= \sum_{j=0}^{\infty} \sum_{s^{t+j}|s^t} \beta^j \pi_{t+j}(s^{t+j}|s^t) \frac{u_c(s^{t+j})}{u_c(s^t)} \; z(s^{t+j}) \\ &= \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} \; z(s^{t+j}) \end{aligned} \tag{7.131}$$

Restrictions (7.131) form a sequence of *implementability constraints* on a Ramsey allocation in an AMSS economy.

Expression (7.131) at time $t = 0$ and initial state s^0 is an *implementability constraint* on a Ramsey allocation in a Lucas-Stokey economy too:

$$b_0(s^{-1}) = \mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} \; z(s^j). \tag{7.132}$$

Comparison with Lucas-Stokey economy

It is instructive to compare the present economy without state-contingent debt to *the Lucas-Stokey economy*.

Suppose that the initial government debt in period 0 and state s^0 is the same across the two economies:

$$b_0(s^{-1}) = b_0(s_0|s^{-1}).$$

Implementability condition (7.132) of our AMSS economy is exactly the same as the one for the Lucas-Stokey economy

While (7.132) is the only implementability condition arising from budget constraints in the complete markets economy, many more implementability conditions, namely (7.131), must be satisfied in the AMSS economy

Because the beginning-of-period indebtedness is the same across any two histories, for any two realizations s^t and $\tilde{s}^t$ that share the same history until the previous period, i.e., $s^{t-1} = \tilde{s}^{t-1}$, we must impose equality across the right sides of their respective budget constraints, as depicted in expression (7.130).

Ramsey problem without state-contingent debt

The Ramsey planner chooses an allocation that solves

$$\max_{\{c_t(s^t), b_{t+1}(s^t)\}} \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u\left(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)\right)$$

where the maximization is subject to

$$\mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} \; z(s^j) \geq b_0(s^{-1}) \tag{7.133}$$

and

$$\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} \, z(s^{t+j}) = b_t(s^{t-1}) \quad \forall \, s^t \tag{7.134}$$

given $b_0(s^{-1})$

Here we have

- substituted the resource constraint into the utility function, and
- substituted the resource constraint into the net-of-interest government surplus, and
- used the household's first-order condition, $1 - \tau_t^n(s^t) = u_\ell(s^t)/u_c(s^t)$, to eliminate the labor tax rate.

Hence, the net-of-interest government surplus satisfies

$$z(s^t) = \left[1 - \frac{u_\ell(s^t)}{u_c(s^t)}\right] \left[c_t(s^t) + g_t(s_t)\right] - g_t(s_t) - T_t(s^t) \, . \tag{7.135}$$

Lagrangian formulation

Let $\gamma_0(s^0)$ be a nonnegative Lagrange multiplier on constraint (7.133)

As in the Lucas-Stokey economy, this multiplier is strictly positive if the government must resort to distortionary taxation; otherwise it equals zero

A consequence of the assumption that markets in state-contingent securities have been shut down but that a market in a risk-free security remains is that we have to attach stochastic processes $\{\gamma_t(s^t)\}_{t=1}^{\infty}$ of Lagrange multipliers to the implementability constraints (7.134)

Depending on how the constraints bind, these multipliers might be positive or negative,

$$\gamma_t(s^t) \;\geq\; (\leq) \; 0 \quad \text{if the constraint binds in this direction}$$
$$\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} \, z(s^{t+j}) \;\geq\; (\leq) \; b_t(s^{t-1}).$$

A negative multiplier $\gamma_t(s^t) < 0$ means that if we could relax constraint , 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

[2] We will soon see from the first-order conditions of the Ramsey problem, there would then exist 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 takes on a positive value $\gamma_t(\tilde{s}^t) > 0$.

Some calculations

Apply two transformations to the Lagrangian.

Multiply constraint by $u_c(s^0)$ and the constraints by $\beta^t u_c(s^t)$.

Then a Lagrangian for the Ramsey problem can be represented as

$$\begin{aligned} J &= \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \Big\{ u\left(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)\right) \\ &\qquad + \gamma_t(s^t) \Big[\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j u_c(s^{t+j})\, z(s^{t+j}) - u_c(s^t)\, b_t(s^{t-1}) \Big] \Big\} \\ &= \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \Big\{ u\left(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)\right) \\ &\qquad + \Psi_t(s^t)\, u_c(s^t)\, z(s^t) - \gamma_t(s^t)\, u_c(s^t)\, b_t(s^{t-1}) \Big] \Big\}, \end{aligned} \tag{7.136}$$

where

$$\Psi_t(s^t) = \Psi_{t-1}(s^{t-1}) + \gamma_t(s^t) \quad \text{and} \quad \Psi_{-1}(s^{-1}) = 0. \tag{7.137}$$

In (7.136), the second equality uses the law of iterated expectations and Abel's summation formula.

The first-order condition with respect to $c_t(s^t)$ can be expressed as

$$\begin{aligned} u_c(s^t) - u_\ell(s^t) + \Psi_t(s^t) \left\{ \left[u_{cc}(s^t) - u_{c\ell}(s^t) \right] z(s^t) + u_c(s^t)\, z_c(s^t) \right\} \\ - \gamma_t(s^t) \left[u_{cc}(s^t) - u_{c\ell}(s^t) \right] b_t(s^{t-1}) = 0 \end{aligned} \tag{7.138}$$

and with respect to $b_t(s^t)$ as

$$\mathbb{E}_t \left[\gamma_{t+1}(s^{t+1})\, u_c(s^{t+1}) \right] = 0 \tag{7.139}$$

If we substitute $z(s^t)$ from (7.135) and its derivative $z_c(s^t)$ into first-order condition (7.138), we find only 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 (7.138) does not appear in the corresponding expression for the Lucas-Stokey economy
   - This term reflects the constraint that beginning-of-period government indebtedness must be the same across all realizations of next period's state, a constraint that is not present if government debt can be state contingent.
2. The Lagrange multiplier $\Psi_t(s^t)$ in first-order condition (7.138) may change over time in response to realizations of the state, while the multiplier Φ in the Lucas-Stokey economy is time invariant.

To analyze the AMSS model, we find it useful to adopt a recursive formulation using techniques like those in the lectures on *dynamic stackelberg models* and *optimal taxation with state-contingent debt*

A recursive version of AMSS model

We now describe a recursive version 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 (7.132) from the Lucas-Stokey economy
- while adding measurability constraints (7.131) on functions of tails of the 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

$$\tau_t(s^t)n_t(s^t) + T_t(s^t) + b_{t+1}(s^t)/R_t(s^t) = g_t + b_t(s^{t-1}),$$

where $R_t(s^t)$ is the gross risk-free rate of interest between t and $t+1$ at history s^t and $T_t(s^t)$ are nonnegative transfers.

In most of the remainder of this lecture, we shall set transfers to zero.

In this case, the household faces a sequence of budget constraints

$$b_t(s^{t-1}) + (1 - \tau_t(s^t))n_t(s^t) = c_t(s^t) + b_{t+1}(s^t)/R_t(s^t). \tag{7.140}$$

The household's first-order conditions are $u_{c,t} = \beta R_t \mathbb{E}_t u_{c,t+1}$ and $(1 - \tau_t)u_{c,t} = u_{l,t}$.

Using these to eliminate R_t and τ_t from the budget constraint (7.140) gives

$$b_t(s^{t-1}) + \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)}n_t(s^t) = c_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t)}{u_{c,t}(s^t)} \tag{7.141}$$

or

$$u_{c,t}(s^t)b_t(s^{t-1}) + u_{l,t}(s^t)n_t(s^t) = u_{c,t}(s^t)c_t(s^t) + \beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t) \tag{7.142}$$

Now define

$$x_t \equiv \beta b_{t+1}(s^t)\mathbb{E}_t u_{c,t+1} = u_{c,t}(s^t)\frac{b_{t+1}(s^t)}{R_t(s^t)} \tag{7.143}$$

and represent the household's budget constraint at time t, history s^t as

$$\frac{u_{c,t}x_{t-1}}{\beta \mathbb{E}_{t-1} u_{c,t}} = u_{c,t}c_t - u_{l,t}n_t + x_t \tag{7.144}$$

for $t \geq 1$.

Measurability constraints

Write equation (7.142) as

$$b_t(s^{t-1}) = c_t(s^t) - \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)} n_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1}) b_{t+1}(s^t)}{u_{c,t}}, \tag{7.145}$$

The right side of equation (7.145) 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 (7.145) must equal $b_t(s^{t-1})$

The means that is has to be *measurable* with respect to s^{t-1}

These are the *measureablility 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

$$V(x_-, s_-) = \max_{\{n(s), x(s)\}} \sum_s \Pi(s|s_-) \left[u(n(s) - g(s), 1 - n(s)) + \beta V(x(s), s)\right] \tag{7.146}$$

subject to the following collection of implementability constraints, one for each $s \in \mathcal{S}$:

$$\frac{u_c(s) x_-}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_c(\tilde{s})} = u_c(s)(n(s) - g(s)) - u_l(s) n(s) + x(s). \tag{7.147}$$

A continuation Ramsey planner at $t \geq 1$ takes $(x_{t-1}, s_{t-1}) = (x_-, s_-)$ as given and chooses $(n_t(s_t), x_t(s_t)) = (n(s), x(s))$ for $s \in \mathcal{S}$ before s_t is realized.

The **Ramsey planner** takes (b_0, s_0) as given and chooses (n_0, x_0).

The Bellman equation for the time $t = 0$ Ramsey planner is

$$W(b_0, s_0) = \max_{n_0, x_0} u(n_0 - g_0, 1 - n_0) + \beta V(x_0, s_0) \tag{7.148}$$

subject to

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + x_0. \tag{7.149}$$

Martingale supercedes state-variable degeneracy

Let $\mu(s|s_-)\Pi(s|s_-)$ be a Lagrange multiplier on constraint (7.147) for state s.

After forming an appropriate Lagrangian, we find that the continuation Ramsey planner's first-order condition with respect to $x(s)$ is

$$\beta V_x(x(s), s) = \mu(s|s_-). \tag{7.150}$$

Applying the envelope theorem to Bellman equation (7.146) gives

$$V_x(x_-, s_-) = \sum_s \Pi(s|s_-)\mu(s|s_-)\frac{u_c(s)}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s_-)u_c(\tilde{s})}. \tag{7.151}$$

Equations (7.150) and (7.151) imply that

$$V_x(x_-, s_-) = \sum_s \left(\Pi(s|s_-)\frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}|s_-)u_c(\tilde{s})}\right) V_x(x(s), s). \tag{7.152}$$

Equation (7.152) states that $V_x(x, s)$ is a *risk-adjusted martingale*

In saying that math:*V_x(x, s)* is a risk-adjusted martingale we mean that $V_x(x, s)$ is a martingale with respect to the probability distribution over s^t sequences generated by the *twisted* transition probability matrix:

$$\check{\Pi}(s|s_-) \equiv \Pi(s|s_-)\frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}|s_-)u_c(\tilde{s})}.$$

Nonnegative transfers

Suppose that instead of imposing $T_t = 0$, we impose a nonnegativity constraint $T_t \geq 0$ on transfers

We also consider the special case of quasi-linear preferences studied by AMSS, $u(c, l) = c + H(l)$.

In this case, $V_x(x, s) \leq 0$ is a non-positive martingale.

By the *martingale convergence theorem* $V_x(x, s)$ converges almost surely.

Furthermore, when the Markov chain $\Pi(s|s_-)$ and the government expenditure function $g(s)$ are such that g_t is perpetually random, $V_x(x, s)$ almost surely converges to zero.

For quasi-linear preferences, the first-order condition with respect to $n(s)$ becomes

$$(1 - \mu(s|s_-))(1 - u_l(s)) + \mu(s|s_-)n(s)u_{ll}(s) = 0.$$

Since $\mu(s|s_-) = \beta V_x(x(s), x)$ converges to zero, in the limit $u_l(s) = 1 = u_c(s)$, so that $\tau(x(s), s) = 0$.

Thus, in the limit, 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.

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 also the initial government debt b_0.

In *our recursive formulation of the Lucas-Stokey*, we found that

- the counterpart to $V_x(x, s)$ is time invariant and equal to the Lagrange multiplier on the Lucas-Stokey implementability constraint
- the 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 is what transmits a twisted martingale-like component to consumption, employment, and the tax rate

Special case of AMSS model

That the Ramsey allocation for the AMSS model differs from the Ramsey allocation of the Lucas-Stokey model is a symptom that the measurability constraints (7.131) bind

Following Bhandari, Evans, Golosov, and Sargent *[BEGS13]* (henceforth BEGS), we now consider a special case of the AMSS model in which these constraints don't bind.

Here the AMSS Ramsey planner chooses not to issue state-contingent debt, though he is free to do so.

The environment is one in which fluctuations in the risk-free interest rate provide the insurance that the Ramsey planner wants.

Following BEGS, we set $S = 2$ and assume that the state s_t is i.i.d., so that the transition matrix $\Pi(s'|s) = \Pi(s')$ for $s = 1, 2$.

Following BEGS, it is useful to consider the following special case of the implementability constraints evaluated at the constant value of the state variable $x_- = x(s) = \check{x}$:

$$\frac{u_c(s)\check{x}}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}) u_c(\tilde{s})} = u_c(s)(n(s) - g(s)) - u_l(s)n(s) + \check{x}, \quad s = 1, 2. \tag{7.153}$$

We guess and verify that the scaled Lagrange multiplier $\mu(s) = \mu$ is a constant that is independent of s.

At a fixed x, because $V_x(x, s)$ must be independent of s_-, the risk-adjusted martingale equation (7.152) becomes

$$V_x(\check{x}) = \sum_s \left(\Pi(s) \frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}) u_c(\tilde{s})} \right) V_x(\check{x}) = V_x(\check{x}).$$

This confirms that $\mu(s) = \beta V_x$ is constant at some value μ.

For the continuation Ramsey planner facing implementability constraints (7.153), the first-order conditions with respect to $n(s)$ become

$$u_c(s) - u_l(s)\mu \times \left\{ \frac{\check{x}}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}) u_c(\tilde{s})} (u_{cc}(s) - u_{cl}(s)) - u_c(s) - n(s)(u_{cc}(s) - u_{cl}(s)) - u_{lc}(s) n(s) + u_l(s) \right\} = 0. \tag{7.154}$$

Equations (7.154) and are four equations in the four unknowns $n(s), s = 1, 2$, $\check{x}$, and μ.

Under some regularity conditions on the period utility function $u(c, l)$, BEGS show that these equations have a unique solution featuring a negative value of $\check{x}$.

Consumption $c(s)$ and the flat-rate tax on labor $\tau(s)$ can then be constructed as history-independent functions of s.

In this AMSS economy, $\check{x} = x(s) = u_c(s) \frac{b_{t+1}(s)}{R_t(s)}$.

The risk-free interest rate, the tax rate, and the marginal utility of consumption fluctuate with s, but x does not and neither does $\mu(s)$.

The labor tax rate and the allocation depend only on the current value of s.

For this special AMSS economy to be in a steady state from time 0 onward, it is necessary that initial debt b_0 satisfy the time 0 implementability constraint at the value $\check{x}$ and the realized value of s_0.

We can solve for this level of b_0 by plugging the $n(s_0)$ and $\check{x}$ that solve our four equation system into

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} n_0 + \check{x}$$

and solving for b_0.

This b_0 assures that a steady state $\check{x}$ prevails from time 0 on.

Relationship to a Lucas-Stokey economy

The constant value of the Lagrange multiplier $\mu(s)$ in the Ramsey plan for our special AMSS economy is a tell tale sign that the measurability restrictions imposed on the Ramsey allocation by the requirement that government debt must be risk free are slack.

When they bind, those measurability restrictions cause the AMSS tax policy and allocation to be history dependent — that's what activates flucations in the risk-adjusted martingale.

Because those measurability conditions are slack in this special AMSS economy, we can also view this as a Lucas-Stokey economy that starts from a particular initial government debt.

The setting of b_0 for the corresponding Lucas-Stokey implementability constraint solves

$$u_{c,0} b_0 = u_{c,0}(n_0 - g_0) - u_{l,0} + \beta \check{x}.$$

In this Lucas-Stokey economy, although the Ramsey planner is free to issue state-contingent debt, it chooses not to and instead issues only risk-free debt.

It achieves the risk-sharing it wants with the private sector by altering the amounts of one-period risk-free debt it issues at each current state, while understanding the equilibrium interest rate fluctuations that its tax policy induces.

Convergence to the special case

In an i.i.d., $S = 2$ AMSS economy in which the initial b_0 does not equal the special value described in the previous subsection, x fluctuates and is history-dependent.

The Lagrange multiplier $\mu_s(s^t)$ is a non trivial risk-adjusted martingale and the allocation and distorting tax rate are both history dependent, as is true generally in an AMSS economy.

However, BEGS describe conditions under which such an i.i.d., $S = 2$ AMSS economy in which the representative agent dislikes consumption risk converges to a Lucas-Stokey economy in the sense that $x_t \to \check{x}$ as $t \to \infty$.

The following subsection displays a numerical example that exhibits convergence.

Examples

We now turn to some computations

Code to generate the following figures can be found in the files amss.py and amss_figures.py

Before running the code, save lucas_stokey.py, CES.py, and BGP.py in your present working directory

```
# -*- coding: utf-8 -*-
"""
Created on Sun Feb 22 10:47:42 2015

@author: dgevans
"""
import numpy as np
from scipy.interpolate import UnivariateSpline

class interpolate_wrapper(object):
    '''
    Wrapper to interpolate vector function
    '''
    def __init__(self,F):
        '''
        Inits with array of interpolated functions
        '''
        self.F = F

    def __getitem__(self,index):
        '''
        Uses square brakets operator
        '''
        return interpolate_wrapper(np.asarray(self.F[index]))
```

```
    def reshape(self,*args):
        '''
        Reshapes F
        '''
        self.F = self.F.reshape(*args)
        return self

    def transpose(self):
        '''
        Transpose F
        '''
        self.F = self.F.transpose()

    def __len__(self):
        '''
        return length
        '''
        return len(self.F)

    def __call__(self,xvec):
        '''
        Evaluates F at X for each element of F, keeping track of the shape of␣
↪F
        '''
        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(object):
    '''
    Generates an interpolator factory which will interpolate vector functions
    '''
    def __init__(self,k,s):
        '''
        Inits with types, orders and k
        '''
        self.k = k
        self.s = s

    def __call__(self,xgrid,Fs):
        '''
        Interpolates function given function values Fs at domain X
        '''
        shape,m = Fs.shape[:-1],Fs.shape[-1]
        Fs = Fs.reshape((-1,m))
        F = []
        for Fhat in Fs:
            #F.append(interpolate(X,Fs[:,i],self.INFO))
```

```
            F.append(UnivariateSpline(xgrid,Fhat,k=self.k,s=self.s))
        return interpolate_wrapper(np.array(F).reshape(shape))


def fun_vstack(fun_list):
    '''
    Performs vstack on interpolator wrapper
    '''
    Fs = [IW.F for IW in fun_list]
    return interpolate_wrapper(np.vstack(Fs))

def fun_hstack(fun_list):
    '''
    Performs vstack on interpolator wrapper
    '''
    Fs = [IW.F for IW in fun_list]
    return interpolate_wrapper(np.hstack(Fs))

def simulate_markov(Pi,s_0,T):
    '''
    Simulates markov chain Pi for T periods starting at s_0
    '''

    sHist = np.empty(T,dtype = int)
    sHist[0] = s_0
    S = len(Pi)
    for t in range(1,T):
        sHist[t] = np.random.choice(np.arange(S),p=Pi[sHist[t-1]])

    return sHist
```

```
# -*- coding: utf-8 -*-
"""
Created on Sun Feb 22 10:26:11 2015

@author: dgevans
"""
import numpy as np
from scipy.optimize import root
from scipy.optimize import fmin_slsqp
import lucas_stokey as LS

class Planners_Allocation_Bellman(object):
    '''
    Compute the planner's allocation by solving Bellman
    equation.
    '''
    def __init__(self,Para,mugrid):
        '''
        Initializes the class from the calibration Para
        '''
        self.beta = Para.beta
        self.Pi = Para.Pi
```

```
        self.G = Para.G
        self.S = len(Para.Pi) # number of states
        self.Theta = Para.Theta
        self.Para = Para
        self.mugrid = mugrid

        #now 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 Para and initial␣
↪grid mugrid0
        '''
        Para,mugrid0 = self.Para,self.mugrid
        Pi = Para.Pi
        S = len(Para.Pi)

        #First get initial fit from lucas stockey solution.
        #Need to change things to be ex_ante
        PP = LS.Planners_Allocation_Sequential(Para)
        interp = interpolator_factory(2,None)

        def incomplete_allocation(mu_,s_):
            c,n,x,V = PP.time1_value(mu_)
            return c,n,Pi[s_].dot(x),Pi[s_].dot(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
            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(mugrid0))
        self.xgrid = xgrid

        #Now iterate on Bellman equation
        T = BellmanEquation(Para,xgrid,policies)
        diff = 1.
```

```
        while diff > 1e-6:
            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 Tau(self,c,n):
        '''
        Computes Tau given c,n
        '''
        Para = self.Para
        Uc,Un = Para.Uc(c,n),Para.Un(c,n)

        return 1+Un/(self.Theta * 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
        '''
        Para,Pi = self.Para,self.Pi
        Uc = Para.Uc
        cf,nf,xprimef,Tf = self.policies

        if sHist == None:
            sHist = simulate_markov(Pi,s_0,T)

        cHist,nHist,Bhist,xHist,TauHist,THist,muHist = np.zeros((7,T))
        #time0
        cHist[0],nHist[0],xHist[0],THist[0]  = self.time0_allocation(B_,s_0)
        TauHist[0] = self.Tau(cHist[0],nHist[0])[s_0]
        Bhist[0] = B_
        muHist[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)

            Tau = self.Tau(c,n)[s]
            u_c = Uc(c,n)
            Eu_c = Pi[s_,:].dot(u_c)

            muHist[t] = self.Vf[s](xprime[s])

            cHist[t],nHist[t],Bhist[t],TauHist[t] = c[s],n[s],x/Eu_c,Tau
            xHist[t],THist[t] = xprime[s],T[s]
        return cHist,nHist,Bhist,xHist,TauHist,THist,muHist,sHist


class BellmanEquation(object):
    '''
    Bellman equation for the continuation of the Lucas-Stokey Problem
    '''
    def __init__(self,Para,xgrid,policies0):
        '''
        Initializes the class from the calibration Para
        '''
        self.beta = Para.beta
        self.Pi = Para.Pi
        self.G = Para.G
        self.S = len(Para.Pi) # number of states
        self.Theta = Para.Theta
        self.Para = Para

        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
        '''
        Para = self.Para
        S,Theta,Uc,Un,G = self.S,self.Theta,Para.Uc,Para.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].
↪dot(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:
            PF = lambda x,s: self.get_policies_time1(x,s,Vf)
        else:
            PF = lambda B_,s0: self.get_policies_time0(B_,s0,Vf)
        return PF

    def get_policies_time1(self,x,s_,Vf):
```

```
        '''
        Finds the optimal policies
        '''
        Para,beta,Theta,G,S,Pi = self.Para,self.beta,self.Theta,self.G,self.S,
↪self.Pi
        U,Uc,Un = Para.U,Para.Uc,Para.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_].dot(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_].dot(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 Para.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)

        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
        '''
        Para,beta,Theta,G = self.Para,self.beta,self.Theta,self.G
        U,Uc,Un = Para.U,Para.Uc,Para.Un

        def objf(z):
            c,n,xprime = z[:-1]

            return -(U(c,n)+beta*Vf[s0](xprime))

        def cons(z):
            c,n,xprime,T = z
            return np.hstack([
```

```
            -Uc(c,n)*(c-B_-T)-Un(c,n)*n - beta*xprime,
            (Theta*n - c - G)[s0]
            ])

        if Para.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])
```

```
# -*- coding: utf-8 -*-
"""
Created on Fri Feb 20 14:07:56 2015

@author: dgevans
"""
import matplotlib.pyplot as plt
import numpy as np
import lucas_stokey as LS
from BGP import M1
from CES import M1 as M_convergence
from CES import M_time_example

#initialize mugrid for value function iteration
muvec = np.linspace(-0.7,0.01,200)


'''
Time Varying Example
'''

M_time_example.transfers = True #Government can use transfers
PP_seq_time = LS.Planners_Allocation_Sequential(M_time_example) #solve␣
↪sequential problem
PP_im_time = Planners_Allocation_Bellman(M_time_example,muvec)

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 = PP_seq_time.simulate(1.,0,7,sHist_h)
sim_im_h = PP_im_time.simulate(1.,0,7,sHist_h)
sim_seq_l = PP_seq_time.simulate(1.,0,7,sHist_l)
sim_im_l = PP_im_time.simulate(1.,0,7,sHist_l)

p1=plt.figure(figsize=[14,10])
plt.subplot(3,2,1)
plt.title('Consumption')
```

```
plt.plot(sim_seq_l[0],'-ok')
plt.plot(sim_im_l[0],'-or')
plt.plot(sim_seq_h[0],'-^k')
plt.plot(sim_im_h[0],'-^r')
plt.subplot(3,2,2)
plt.title('Labor')
plt.plot(sim_seq_l[1],'-ok')
plt.plot(sim_im_l[1],'-or')
plt.plot(sim_seq_h[1],'-^k')
plt.plot(sim_im_h[1],'-^r')
plt.legend(('Complete Markets','Incomplete Markets'),loc='best')
plt.subplot(3,2,3)
plt.title('Government Debt')
plt.plot(sim_seq_l[2],'-ok')
plt.plot(sim_im_l[2],'-or')
plt.plot(sim_seq_h[2],'-^k')
plt.plot(sim_im_h[2],'-^r')
plt.subplot(3,2,4)
plt.title('Tax Rate')
plt.plot(sim_seq_l[3],'-ok')
plt.plot(sim_im_l[4],'-or')
plt.plot(sim_seq_h[3],'-^k')
plt.plot(sim_im_h[4],'-^r')
plt.subplot(3,2,5)
plt.title('Government Spending')
plt.plot(M_time_example.G[sHist_l],'-ok')
plt.plot(M_time_example.G[sHist_l],'-or')
plt.plot(M_time_example.G[sHist_h],'-^k')
plt.plot(M_time_example.G[sHist_h],'-^r')
plt.ylim([0.05,0.25])
plt.subplot(3,2,6)
plt.title('Output')
plt.plot(M_time_example.Theta[sHist_l]*sim_seq_l[1],'-ok')
plt.plot(M_time_example.Theta[sHist_l]*sim_im_l[1],'-or')
plt.plot(M_time_example.Theta[sHist_h]*sim_seq_h[1],'-^k')
plt.plot(M_time_example.Theta[sHist_h]*sim_im_h[1],'-^r')
plt.tight_layout()
plt.savefig('TaxSequence_time_varying_AMSS.png')




'''
BGP Example
'''

M1.transfers = False #Government can use transfers
PP_seq = LS.Planners_Allocation_Sequential(M1) #solve sequential problem
PP_bel = LS.Planners_Allocation_Bellman(M1,muvec) #solve recursive problem
PP_im = Planners_Allocation_Bellman(M1,muvec)

T = 20
#sHist = utilities.simulate_markov(M1.Pi,0,T)
```

```
sHist = np.array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0],
↪dtype=int)

#simulate
sim_seq = PP_seq.simulate(0.5,0,T,sHist)
#sim_bel = PP_bel.simulate(0.5,0,T,sHist)
sim_im = PP_im.simulate(0.5,0,T,sHist)

#plot policies
p2=plt.figure(figsize=[14,10])
plt.subplot(3,2,1)
plt.title('Consumption')
plt.plot(sim_seq[0],'-ok')
#plt.plot(sim_bel[0],'-xk')
plt.plot(sim_im[0],'-^k')
plt.legend(('Complete Markets','Incomplete Markets'),loc='best')
plt.subplot(3,2,2)
plt.title('Labor')
plt.plot(sim_seq[1],'-ok')
#plt.plot(sim_bel[1],'-xk')
plt.plot(sim_im[1],'-^k')
plt.subplot(3,2,3)
plt.title('Government Debt')
plt.plot(sim_seq[2],'-ok')
#plt.plot(sim_bel[2],'-xk')
plt.plot(sim_im[2],'-^k')
plt.subplot(3,2,4)
plt.title('Tax Rate')
plt.plot(sim_seq[3],'-ok')
#plt.plot(sim_bel[3],'-xk')
plt.plot(sim_im[4],'-^k')
plt.subplot(3,2,5)
plt.title('Government Spending')
plt.plot(M1.G[sHist],'-ok')
#plt.plot(M1.G[sHist],'-^k')
plt.ylim([0.05,0.25])
plt.subplot(3,2,6)
plt.title('Output')
plt.plot(M1.Theta[sHist]*sim_seq[1],'-ok')
#plt.plot(M1.Theta[sHist]*sim_bel[1],'-xk')
plt.plot(M1.Theta[sHist]*sim_im[1],'-^k')
plt.savefig('TaxSequence_AMSS.png')
plt.tight_layout()


#Now long simulations
T_long = 200
sim_seq_long = PP_seq.simulate(0.5,0,T_long)
sHist_long = sim_seq_long[-3]
sim_im_long = PP_im.simulate(0.5,0,T_long,sHist_long)

p3=plt.figure(figsize=[14,10])
plt.subplot(3,2,1)
```

```
plt.title('Consumption')
plt.plot(sim_seq_long[0],'-k')
plt.plot(sim_im_long[0],'-.k')
plt.legend(('Complete Markets','Incomplete Markets'),loc='best')
plt.subplot(3,2,2)
plt.title('Labor')
plt.plot(sim_seq_long[1],'-k')
plt.plot(sim_im_long[1],'-.k')
plt.subplot(3,2,3)
plt.title('Government Debt')
plt.plot(sim_seq_long[2],'-k')
plt.plot(sim_im_long[2],'-.k')
plt.subplot(3,2,4)
plt.title('Tax Rate')
plt.plot(sim_seq_long[3],'-k')
plt.plot(sim_im_long[4],'-.k')
plt.subplot(3,2,5)
plt.title('Government Spending')
plt.plot(M1.G[sHist_long],'-k')
plt.plot(M1.G[sHist_long],'-.k')
plt.ylim([0.05,0.25])
plt.subplot(3,2,6)
plt.title('Output')
plt.plot(M1.Theta[sHist_long]*sim_seq_long[1],'-k')
plt.plot(M1.Theta[sHist_long]*sim_im_long[1],'-.k')
plt.tight_layout()
plt.savefig('Long_SimulationAMSS.png')

'''
Show Convergence example
'''
muvec = np.linspace(-0.15,0.0,100) #change
PP_C = Planners_Allocation_Bellman(M_convergence,muvec)
xgrid = PP_C.xgrid
xf = PP_C.policies[-2] #get x policies
plt.figure()
xprimes0=[]
xprimes1=[]
for x in xgrid:
    xprimes0.append(xf[0,0](x))
    xprimes1.append(xf[0,1](x))

xprimes0=np.hstack(xprimes0)
xprimes1=np.hstack(xprimes1)
p4=plt.figure()
plt.plot(xgrid,xprimes0-xgrid)
plt.plot(xgrid,xprimes1-xgrid)

sim_seq_convergence = PP_C.simulate(0.5,0,2000)
sHist_long = sim_seq_convergence[-1]

p5=plt.figure(figsize=[14,10])
plt.subplot(3,2,1)
```

```
plt.title('Consumption')
plt.plot(sim_seq_convergence[0],'-k')
plt.legend(('Complete Markets','Incomplete Markets'),loc='best')
plt.subplot(3,2,2)
plt.title('Labor')
plt.plot(sim_seq_convergence[1],'-k')
plt.subplot(3,2,3)
plt.title('Government Debt')
plt.plot(sim_seq_convergence[2],'-k')
plt.subplot(3,2,4)
plt.title('Tax Rate')
plt.plot(sim_seq_convergence[3],'-k')
plt.subplot(3,2,5)
plt.title('Government Spending')
plt.plot(M_convergence.G[sHist_long],'-k')
plt.ylim([0.05,0.25])
plt.subplot(3,2,6)
plt.title('Output')
plt.plot(M_convergence.Theta[sHist_long]*sim_seq_convergence[1],'-k')
plt.tight_layout()
plt.savefig('Convergence_SimulationAMSS.png')
```

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 terms of matching our computer code, we have expressed utility as a function of n rather than leisure l

As before, we set $\gamma = \sigma = 2$ and $\beta = .9$

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 the components of the state vector as the following six (t, g) pairs:

$$(0, g_l), (1, g_l), (2, g_l), (3, g_l), (3, g_h), (t \geq 4, g_l).$$

We think of these 6 states as corresponding to $s = 1, 2, 3, 4, 5, 6$

The transition matrix is

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

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 the *Lucas-Stokey economy*

The following figure plots the Ramsey tax policy under both complete and incomplete markets for both possible realizations of the state at time $t = 3$.

The optimal policies with access to state contingent debt are represented by black lines, while the optimal policies with a risk free bond are in red.

Paths with circles are histories in which there is peace, while those with triangle denote war

```
p1
```

How the 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 $t = 2$, it increases its debt burden in the states when there is peace

That helps it to reduce the debt burden when there is war

This pattern facilities smoothing tax rates across states

The government without state contingent debt cannot do this

Instead, it must enter with the same level of debt at all possible states of the world at time $t = 3$.

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 policy 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 where the government perpetually faces the prospect of war

This case was studied in the final example of the lecture on *optimal taxation with state-contingent debt*

There, each period the government faces a constant probability, 0.5, of war.

In addition, this example features the following preferences

$$u(c, n) = \log(c) + 0.69 \log(1 - n)$$

With these preferences, Ramsey tax rates will vary even in the Lucas-Stokey model with state-contingent debt

The figure below plots the optimal tax policies for both the case of state contingent (circles) and the case of non state contingent debt (triangles).

```
p2
```

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 must finance 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 received.

This is even more evident in the following figure that plots the evolution of the two policies over 200 periods.

```
p3
```

Default Risk and Income Fluctuations

Contents

- *Default Risk and Income Fluctuations*
  - *Overview*
  - *Structure*

- *Equilibrium*
- *Computation*
- *Results*
- *Exercises*
- *Solutions*

Overview

This lecture computes versions of Arellano's *[Are08]* model of sovereign default

The model describes interactions among default risk, output, and an equilibrium interest rate that includes a premium for endogenous default risk

The decision maker is a government of a small open economy that borrows from risk-neutral foreign creditors

The foreign lenders must be compensated for default risk

The government borrows and lends abroad in order to smooth the consumption of its citizens

The government repays its debt only if it wants to, but declining to pay has adverse consequences

The interest rate on government debt adjusts in response to the state-dependent default probability chosen by government

The model yields outcomes that help interpret sovereign default experiences, including

- countercyclical interest rates on sovereign debt
- countercyclical trade balances
- high volatility of consumption relative to output

Notably, long recessions caused by bad draws in the income process increase the government's incentive to default

This can lead to

- spikes in interest rates
- temporary losses of access to international credit markets
- large drops in output, consumption, and welfare
- large capital outflows during recessions

Such dynamics are consistent with experiences of many countries

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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{7.155}$$

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 (7.155)

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

$$c = y + B - q(B', y)B' \tag{7.156}$$

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 δ, the expected value of a promise to pay one unit of consumption next period is $1 - \delta$.

Therefore, the discounted expected value of a promise to pay B next period is

$$q = \frac{1-\delta}{1+r} \tag{7.157}$$

Next we turn to how the government in effect chooses the default probability δ

Government's decisions

At each point in time t, the government chooses between

1. defaulting
2. meeting its current obligations and purchasing or selling an optimal quantity of one-period sovereign debt

Defaulting means declining to repay all of its current obligations

If the government defaults in the current period, then consumption equals current output

But a sovereign default has two consequences:

1. Output immediately falls from y to $h(y)$, where $0 \leq h(y) \leq y$
    - it returns to y only after the country regains access to international credit markets
2. The country loses access to foreign credit markets

Reentering international credit market

While in a state of default, the economy regains access to foreign credit in each subsequent period with probability θ

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 period's debt due to $-B'$

In a recursive formulation,

- state variables for the government comprise the pair (B, y)
- $v(B, y)$ is the optimum value of the government's problem when at the beginning of a period it faces the choice of whether to honor or default
- $v_c(B, y)$ is the value of choosing to pay obligations falling due
- $v_d(y)$ is the value of choosing to default

$v_d(y)$ does not depend on B because, when access to credit is eventually regained, net foreign assets equal 0

Expressed recursively, the value of defaulting is

$$v_d(y) = u(h(y)) + \beta \int \left\{ \theta v(0, y') + (1 - \theta) v_d(y') \right\} p(y, y') dy'$$

The value of paying is

$$v_c(B, y) = \max_{B' \geq -Z} \left\{ u(y - q(B', y)B' + B) + \beta \int v(B', y') p(y, y') dy' \right\}$$

The three value functions are linked by

$$v(B, y) = \max\{v_c(B, y), v_d(y)\}$$

The government chooses to default when

$$v_c(B, y) < v_d(y)$$

and hence given B' the probability of default next period is

$$\delta(B', y) := \int \mathbb{1}\{v_c(B', y') < v_d(y')\} p(y, y') dy' \tag{7.158}$$

Given zero profits for foreign creditors in equilibrium, we can combine (7.157) and (7.158) to pin down the bond price function:

$$q(B', y) = \frac{1 - \delta(B', y)}{1 + r} \tag{7.159}$$

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 decsion rule attain the optimal value function $v(B, y)$, and
- The price function $q(B', y)$ satisfies equation (7.159)

Computation

Let's now compute an equilibrium of Arellano's model

The equilibrium objects are the value function $v(B, y)$, the associated default decision rule, and the pricing function $q(B', y)$

We'll use our code to replicate Arellano's results

After that we'll perform some additional simulations

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 Tauchen's 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)

```
"""
Filename: arellano_vfi.py

Authors: Chase Coleman, John Stachurski

Solve the Arellano default risk model

References
----------

http://quant-econ.net/py/arellano.html

.. Arellano, Cristina. "Default risk and income fluctuations in emerging
   economies." The American Economic Review (2008): 690-712.

"""
from __future__ import division
import numpy as np
import random
import quantecon as qe
from numba import jit


class Arellano_Economy(object):
    """
    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
    ny : int
        Number of points in y grid
    nB : int
        Number of points in B grid
    tol : float
        Error tolerance in iteration
    maxit : int
        Maximum number of iterations
    """

    def __init__(self,
            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):

        # Save parameters
        self.beta, self.gamma, self.r = beta, gamma, r
        self.rho, self.eta, self.theta = rho, eta, theta
        self.ny, self.nB = ny, nB

        # 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

        # 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 = (np.dot(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.beta, self.theta, self.gamma)

            # Update prices
            Vd_compat = np.repeat(self.Vd, self.nB).reshape(self.ny, self.nB)
            default_states = Vd_compat > self.Vc
            self.default_prob[:, :] = np.dot(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))
            self.V[:, :] = V_upd[:, :]

            it += 1
            if it % 25 == 0:
                print("Running iteration {} with dist of {}".format(it, 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 = np.dot(self.Py, self.V)

        _compute_savings_policy(self.ygrid, self.Bgrid, self.Q, EV,
                self.gamma, self.beta, 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
        # Start off not in default
        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.theta:
                    in_default = False
            B_sim_indices[t+1] = Bi_next
            q_sim[t] = self.Q[yi, 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, gamma):
    return c**(1-gamma)/(1-gamma)


@jit(nopython=True)
def _inner_loop(ygrid, def_y, Bgrid, Vd, Vc, EVc,
                          EVd, EV, qq, beta, theta, gamma):
    """
    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], gamma) + \
                beta * (theta * EVc[iy, zero_ind] + (1 - theta) * 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, gamma) + beta * 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, gamma, beta, 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
```

Results

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

In what follows, all results are computed using Arellano's parameter values

The values can be seen in the *__init__* method of the *Arellano_Economy* shown above

- For example, *r=0.017* matches the average quarterly rate on a 5 year US treasury over the period 1983–2001

Details on how to compute the figures are reported as solutions to the exercises

The first figure shows the bond price schedule and replicates Figure 3 of Arellano, where y_L and Y_H are particular below average and above average values of output y

- y_L is 5% below the mean of the y grid values

- y_H is 5% above the mean of the y grid values

The grid used to compute this figure was relatively coarse (*ny, nB = 21, 251*) in order to match Arrelano's findings

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

In either case, the figure shows that

- Higher levels of debt (larger $-B'$) induce larger discounts on the face value, which correspond to higher interest rates
- Lower income also causes more discounting, as foreign creditors anticipate greater likelihood of default

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 (7.158)

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

Exercises

Exercise 1

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

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

Solutions

Solution notebook

Globalization and Cycles

Contents

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

This lecture is coauthored with Chase Coleman

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

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 other's markets

As illustrated below, this leads to synchonization of business cycles across the two countries

Model

Let's write down the model more formally

(The treatment is relatively terse since full details can be found in the original paper)

Time is discrete with $t = 0, 1, \ldots$

There are two countries indexed by j or k

In each country, a representative household inelastically supplies L_j units of labor at wage rate $w_{j,t}$

Without loss of generality, it is assumed that $L_1 \geq L_2$

Households consume a single nontradeable final good which is produced competitively

Its production involves combining two types of tradeable intermediate inputs via

$$Y_{k,t} = C_{k,t} = \left(\frac{X^o_{k,t}}{1-\alpha}\right)^{1-\alpha} \left(\frac{X_{k,t}}{\alpha}\right)^{\alpha}$$

Here $X^o_{k,t}$ is a homogeneous input which can be produced from labor using a linear, one-for-one technology

It is freely tradeable, competitively supplied, and homogeneous across countries

By choosing the price of this good as numeraire and assuming both countries find it optimal to always produce the homogeneous good, we can set $w_{1,t} = w_{2,t} = 1$

The good $X_{k,t}$ is a composite, built from many differentiated goods via

$$X_{k,t}^{1-\frac{1}{\sigma}} = \int_{\Omega_t} [x_{k,t}(\nu)]^{1-\frac{1}{\sigma}} \, d\nu$$

Here $x_{k,t}(\nu)$ is the total amount of a differentiated good $\nu \in \Omega_t$ that is produced

The parameter $\sigma > 1$ is the direct partial elasticity of substitution between a pair of varieties and Ω_t is the set of varieties available in period t

We can split the varieties into those which are supplied competitively and those supplied monopolistically; that is, $\Omega_t = \Omega_t^c + \Omega_t^m$

Prices

Demand for differentiated inputs is

$$x_{k,t}(\nu) = \left(\frac{p_{k,t}(\nu)}{P_{k,t}}\right)^{-\sigma} \frac{\alpha L_k}{P_{k,t}}$$

Here

- $p_{k,t}(\nu)$ is the price of the variety ν and
- $P_{k,t}$ is the price index for differentiated inputs in k, defined by

$$[P_{k,t}]^{1-\sigma} = \int_{\Omega_t} [p_{k,t}(\nu)]^{1-\sigma} d\nu$$

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 ν produced in country j becomes $p_{k,t}(\nu) = \tau_{j,k}\, p_{j,t}(\nu)$

Using these expressions, we can derive the total demand for each variety, which is

$$D_{j,t}(\nu) = \sum_k \tau_{j,k} x_{k,t}(\nu) = \alpha A_{j,t} (p_{j,t}(\nu))^{-\sigma}$$

where

$$A_{j,t} := \sum_k \frac{\rho_{j,k} L_k}{(P_{k,t})^{1-\sigma}} \quad \text{and} \quad \rho_{j,k} = (\tau_{j,k})^{1-\sigma} \leq 1$$

It is assumed that $\tau_{1,1} = \tau_{2,2} = 1$ and $\tau_{1,2} = \tau_{2,1} = \tau$ for some $\tau > 1$, so that

$$\rho_{1,2} = \rho_{2,1} = \rho := \tau^{1-\sigma} < 1$$

The value $\rho \in [0, 1)$ is a proxy for the degree of globalization

Producing one unit of each differentiated variety requires ψ units of labor, so the marginal cost is equal to ψ for $\nu \in \Omega_{j,t}$

Additionally, all competitive varieties will have the same price (because of equal marginal cost), which means that, for all $\nu \in \Omega^c$,

$$p_{j,t}(\nu) = p^c_{j,t} := \psi \quad \text{and} \quad D_{j,t} = y^c_{j,t} := \alpha A_{j,t} (p^c_{j,t})^{-\sigma}$$

Monopolists will have the same marked-up price, so, for all $\nu \in \Omega^m$,

$$p_{j,t}(\nu) = p^m_{j,t} := \frac{\psi}{1 - \frac{1}{\sigma}} \quad \text{and} \quad D_{j,t} = y^m_{j,t} := \alpha A_{j,t} (p^m_{j,t})^{-\sigma}$$

Define

$$\theta := \frac{p^c_{j,t}}{p^m_{j,t}} \frac{y^c_{j,t}}{y^m_{j,t}} = \left(1 - \frac{1}{\sigma}\right)^{1-\sigma}$$

Using the preceding definitions and some algebra, the price indices can now be rewritten as

$$\left(\frac{P_{k,t}}{\psi}\right)^{1-\sigma} = M_{k,t} + \rho M_{j,t} \quad \text{where} \quad M_{j,t} := N^c_{j,t} + \frac{N^m_{j,t}}{\theta}$$

The symbols $N^c_{j,t}$ and $N^m_{j,t}$ will denote the measures of Ω^c and Ω^m respectively

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 δ as the exogenous probability of a variety becoming obsolete, the dynamic equation for the measure of firms becomes

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

We will work with a normalized measure of varieties

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

We also use $s_j := \frac{L_j}{L_1 + L_2}$ to be the share of labor employed in country j

We can use these definitions and the preceding expressions to obtain a law of motion for $n_t := (n_{1,t}, n_{2,t})$

In particular, given an initial condition, $n_0 = (n_{1,0}, n_{2,0}) \in \mathbb{R}_+^2$, the equilibrium trajectory, $\{n_t\}_{t=0}^\infty = \{(n_{1,t}, n_{2,t})\}_{t=0}^\infty$, is obtained by iterating on $n_{t+1} = F(n_t)$ where $F : \mathbb{R}_+^2 \to \mathbb{R}_+^2$ is given by

$$F(n_t) = \begin{cases} \big(\delta(\theta s_1(\rho) + (1-\theta) n_{1,t}), \delta(\theta s_2(\rho) + (1-\theta) n_{2,t})\big) & \text{for } n_t \in D_{LL} \\ \big(\delta n_{1,t}, \delta n_{2,t}\big) & \text{for } n_t \in D_{HH} \\ \big(\delta n_{1,t}, \delta(\theta h_2(n_{1,t}) + (1-\theta) n_{2,t}\big) & \text{for } n_t \in D_{HL} \\ \big(\delta(\theta h_1(n_{2,t}) + (1-\theta) n_{1,t}, \delta n_{2,t}\big) & \text{for } n_t \in D_{LH} \end{cases}$$

Here

$$\begin{aligned} D_{LL} &:= \{(n_1, n_2) \in \mathbb{R}_+^2 | n_j \leq s_j(\rho)\} \\ D_{HH} &:= \{(n_1, n_2) \in \mathbb{R}_+^2 | n_j \geq h_j(\rho)\} \\ D_{HL} &:= \{(n_1, n_2) \in \mathbb{R}_+^2 | n_1 \geq s_1(\rho) \text{ and } n_2 \leq h_2(n_1)\} \\ D_{LH} &:= \{(n_1, n_2) \in \mathbb{R}_+^2 | n_1 \leq h_1(n_2) \text{ and } n_2 \geq s_2(\rho)\} \end{aligned}$$

while

$$s_1(\rho) = 1 - s_2(\rho) = \min\left\{\frac{s_1 - \rho s_2}{1 - \rho}, 1\right\}$$

and $h_j(n_k)$ is defined implicitly by the equation

$$1 = \frac{s_j}{h_j(n_k) + \rho n_k} + \frac{s_k}{h_j(n_k) + n_k/\rho}$$

Rewriting the equation above gives us a quadratic equation in terms of $h_j(n_k)$

Since we know $h_j(n_k) > 0$ then we can just solve the quadratic equation and return the positive root

This gives us

$$h_j(n_k)^2 + \left((\rho + \frac{1}{\rho})n_k - s_j - s_k\right) h_j(n_k) + (n_k^2 - \frac{s_j n_k}{\rho} - s_k n_k \rho) = 0$$

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

```
"""

Code to simulate the model in Globalization and Synchronization of Innovation
Cycles by Kiminori Matsuyama, Laura Gardini and Iryna Sushko

See <http://www.centreformacroeconomics.ac.uk/Discussion-Papers/2015/
↪CFMDP2015-27-Paper.pdf>

"""


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)
def DHL(n1, n2, s1_rho, s2_rho, s1, s2, theta, delta, rho):
    "Determine whether (n1, n2) is in the set DHL"
    return (n1 >= s1_rho) and (n2 <= _hj(2, n1, s1, s2, theta, delta, rho))

@jit(nopython=True)
def DLH(n1, n2, s1_rho, s2_rho, s1, s2, theta, delta, rho):
    "Determine whether (n1, n2) is in the set DLH"
    return (n1 <= _hj(1, n2, s1, s2, theta, delta, rho)) and (n2 >= s2_rho)

@jit(nopython=True)
def one_step(n1, n2, s1_rho, s2_rho, 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.
    """
    # Depending on where we are, evaluate the right branch
    if DLL(n1, n2, s1_rho, s2_rho, s1, s2, theta, delta, rho):
        n1_tp1 = delta*(theta*s1_rho + (1-theta)*n1)
        n2_tp1 = delta*(theta*s2_rho + (1-theta)*n2)
    elif DHH(n1, n2, s1_rho, s2_rho, s1, s2, theta, delta, rho):
        n1_tp1 = delta*n1
        n2_tp1 = delta*n2
    elif DHL(n1, n2, s1_rho, s2_rho, s1, s2, theta, delta, rho):
```

```
        n1_tp1 = delta*n1
        n2_tp1 = delta*(theta*_hj(2, n1, s1, s2, theta, delta, rho) + (1-
↪theta)*n2)
    elif DLH(n1, n2, s1_rho, s2_rho, s1, s2, theta, delta, rho):
        n1_tp1 = delta*(theta*_hj(1, n2, s1, s2, theta, delta, rho) + (1-
↪theta)*n1)
        n2_tp1 = delta*n2

    return n1_tp1, n2_tp1

@jit(nopython=True)
def n_generator(n1_0, n2_0, s1_rho, s2_rho, s1, s2, theta, delta, rho):
    """
    Given an initial condition, continues to yield new values of
    n1 and n2
    """
    n1_t, n2_t = n1_0, n2_0
    while True:
        n1_tp1, n2_tp1 = one_step(n1_t, n2_t, s1_rho, s2_rho, 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_rho, s2_rho, 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_rho, s2_rho, s1, s2, theta, delta, rho)

    # 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_rho, s2_rho, s1, s2, theta, delta, rho,␣
↪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_rho,␣
↪s2_rho,
                                                        s1, s2, theta, delta,␣
↪rho,
                                                        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
    theta : scalar(Float)
        A measure of how mcuh more of the competitive variety is used in
        production of final goods
    delta : scalar(Float)
        Percentage of firms that are not exogenously destroyed every period
    rho : scalar(Float)
        Measure of how expensive it is to trade between countries
    """
    def __init__(self, s1=0.5, theta=2.5, delta=0.7, rho=0.2):
        # Store model parameters
        self.s1, self.theta, self.delta, self.rho = s1, theta, delta, rho

        # Store other cutoffs and parameters we use
        self.s2 = 1 - s1
        self.s1_rho = self._calc_s1_rho()
        self.s2_rho = 1 - self.s1_rho

    def _unpack_params(self):
        return self.s1, self.s2, self.theta, self.delta, self.rho

    def _calc_s1_rho(self):
        # Unpack params
        s1, s2, theta, delta, rho = self._unpack_params()

        # s_1(rho) = min(val, 1)
        val = (s1 - rho*s2) / (1 - rho)
        return min(val, 1)

    def simulate_n(self, n1_0, n2_0, T):
        """
```

```
        Simulates the values of (n1, n2) for T periods

        Parameters
        ----------
        n1_0 : scalar(Float)
            Initial normalized measure of firms in country one
        n2_0 : scalar(Float)
            Initial normalized measure of firms in country two
        T : scalar(Int)
            Number of periods to simulate

        Returns
        -------
        n1 : Array(Float64, ndim=1)
            A history of normalized measures of firms in country one
        n2 : Array(Float64, ndim=1)
            A history of normalized measures of firms in country two
        """
        # Unpack parameters
        s1, s2, theta, delta, rho = self._unpack_params()
        s1_rho, s2_rho = self.s1_rho, self.s2_rho

        # 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_rho, s2_rho, s1, s2, theta, delta,␣
↪rho)

        # Simulate for T periods
        for t in range(1, T):
            # Get next values
            n1_tp1, n2_tp1 = next(n_gen)

            # Store in arrays
            n1[t] = n1_tp1
            n2[t] = n2_tp1

        return n1, n2

    def pers_till_sync(self, n1_0, n2_0, maxiter=500, npers=3):
        """
        Takes initial values and iterates forward to see whether
        the histories eventually end up in sync.

        If countries are symmetric then as soon as the two countries have the
        same measure of firms then they will 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
        """
        # 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] X [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_rho, s2_rho, s1, s2, theta, delta,
                                      rho, 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

Here's the function

```
def plot_timeseries(n1_0, n2_0, s1=0.5, theta=2.5, delta=0.7, rho=0.2,␣
↪ax=None):
    """
    Plot a single time series with initial conditions
    """
    if ax is None:
        fig, ax = plt.subplots()

    # Create the MSG Model and simulate with initial conditions
    model = MSGSync(s1, theta, delta, rho)
    n1, n2 = model.simulate_n(n1_0, n2_0, 25)

    ax.plot(np.arange(25), n1, label=r"$n_1$", lw=2)
    ax.plot(np.arange(25), n2, label=r"$n_2$", lw=2)

    ax.legend()
    ax.set_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])
plot_timeseries(0.4, 0.3, ax=ax[1])

ax[0].set_title("Not Synchronized")
ax[1].set_title("Synchronized")

fig.tight_layout()

plt.show()
```

Let's 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 ρ translate to more synchronization

You are asked to replicate this figure in the exercises

In the solution to the exercises, you'll also find a figure with sliders, allowing you to experiment with different parameters

Here's one snapshot from the interactive figure

Exercises

Exercise 1

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

Solutions

These solutions are written by Chase Coleman

```
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)
    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()
```

Interactive Version

Additionally, instead of just seeing 4 plots at once, we might want to manually be able to change ρ and see how it affects the plot in real time. Below we use an interactive plot to do this

Note, interactive ploting requires the ipywidgets module to be installed and enabled.

```
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
```

```
    unitrange = np.linspace(0, 1, npts)
    cf = ax.pcolormesh(unitrange, unitrange, ab, cmap="viridis")
    cbar_ax = fig.add_axes([0.95, 0.15, 0.05, 0.7])
    plt.colorbar(cf, cax=cbar_ax)
    plt.show()
    return None
```

```
fig = interact(interact_attraction_basis,
               rho=(0.0, 1.0, 0.05),
               maxiter=(50, 5000, 50),
               npts=(25, 750, 25))
```

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.

BIBLIOGRAPHY

[AJR01] Daron Acemoglu, Simon Johnson, and James A Robinson. The colonial origins of comparative development: an empirical investigation. *The American Economic Review*, 91(5):1369–1401, 2001.

[Aiy94] S Rao Aiyagari. Uninsured Idiosyncratic Risk and Aggregate Saving. *The Quarterly Journal of Economics*, 109(3):659–684, 1994.

[AMSS02] S. Rao Aiyagari, Albert Marcet, Thomas J. Sargent, and Juha Seppala. Optimal Taxation without State-Contingent Debt. *Journal of Political Economy*, 110(6):1220–1254, December 2002.

[AM05] D. B. O. Anderson and J. B. Moore. *Optimal Filtering*. Dover Publications, 2005.

[AHMS96] E. W. Anderson, L. P. Hansen, E. R. McGrattan, and T. J. Sargent. Mechanics of Forming and Estimating Dynamic Linear Economies. In *Handbook of Computational Economics*. Elsevier, vol 1 edition, 1996.

[Are08] Cristina Arellano. Default risk and income fluctuations in emerging economies. *The American Economic Review*, pages 690–712, 2008.

[AP91] Papoulis Athanasios and S Unnikrishna Pillai. *Probability, random variables, and stochastic processes*. Mc-Graw Hill, 1991.

[ACK10] Andrew Atkeson, Varadarajan V Chari, and Patrick J Kehoe. Sophisticated monetary policies*. *The Quarterly journal of economics*, 125(1):47–89, 2010.

[BY04] Ravi Bansal and Amir Yaron. Risks for the Long Run: A Potential Resolution of Asset Pricing Puzzles. *Journal of Finance*, 59(4):1481–1509, 08 2004. URL: https://ideas.repec.org/a/bla/jfinan/v59y2004i4p1481-1509.html, doi:.

[Bar79] Robert J Barro. On the Determination of the Public Debt. *Journal of Political Economy*, 87(5):940–971, 1979.

[Bas05] Marco Bassetto. Equilibrium and government commitment. *Journal of Economic Theory*, 124(1):79–105, 2005.

[BBZ15] Jess Benhabib, Alberto Bisin, and Shenghao Zhu. The wealth distribution in bewley economies with capital income risk. *Journal of Economic Theory*, 159:489–515, 2015.

[BS79] L M Benveniste and J A Scheinkman. On the Differentiability of the Value Function in Dynamic Models of Economics. *Econometrica*, 47(3):727–732, 1979.

[Ber75] Dmitri Bertsekas. *Dynamic Programming and Stochastic Control*. Academic Press, New York, 1975.

[Bew77] Truman Bewley. The permanent income hypothesis: a theoretical formulation. *Journal of Economic Theory*, 16(2):252–292, 1977.

[Bew86] Truman F Bewley. Stationary monetary equilibrium with a continuum of independently fluctuating consumers. In Werner Hildenbran and Andreu Mas-Colell, editors, *Contributions to Mathematical Economics in Honor of Gerard Debreu*, pages 27–102. North-Holland, Amsterdam, 1986.

[BEGS13] Anmol Bhandari, David Evans, Mikhail Golosov, and Thomas J Sargent. Taxes, debts, and redistributions with aggregate shocks. Technical Report, National Bureau of Economic Research, 2013.

[Bis06] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[Cal78] Guillermo A. Calvo. On the time consistency of optimal policy in a monetary economy. *Econometrica*, 46(6):1411–1428, 1978.

[Car01] Christopher D Carroll. A Theory of the Consumption Function, with and without Liquidity Constraints. *Journal of Economic Perspectives*, 15(3):23–45, 2001.

[Car06] Christopher D Carroll. The method of endogenous gridpoints for solving dynamic stochastic optimization problems. *Economics Letters*, 91(3):312–320, 2006.

[Cha98] Roberto Chang. Credible monetary policy in an infinite horizon model: recursive approaches. *Journal of Economic Theory*, 81(2):431–461, 1998.

[CK90] Varadarajan V Chari and Patrick J Kehoe. Sustainable plans. *Journal of Political Economy*, pages 783–802, 1990.

[Col90] Wilbur John Coleman. Solving the Stochastic Growth Model by Policy-Function Iteration. *Journal of Business & Economic Statistics*, 8(1):27–29, 1990.

[CC08] J. D. Cryer and K-S. Chan. *Time Series Analysis*. Springer, 2nd edition edition, 2008.

[DFH06] Steven J Davis, R Jason Faberman, and John Haltiwanger. The flow approach to labor markets: new data sources, micro-macro links and the recent downturn. *Journal of Economic Perspectives*, 2006.

[Dea91] Angus Deaton. Saving and Liquidity Constraints. *Econometrica*, 59(5):1221–1248, 1991.

[DP94] Angus Deaton and Christina Paxson. Intertemporal Choice and Inequality. *Journal of Political Economy*, 102(3):437–467, 1994.

[DH10] Wouter J Den Haan. Comparison of solutions to the incomplete markets model with aggregate uncertainty. *Journal of Economic Dynamics and Control*, 34(1):4–27, 2010.

[DJ92] Raymond J Deneckere and Kenneth L Judd. Cyclical and chaotic behavior in a dynamic equilibrium model, with implications for fiscal policy. *Cycles and chaos in economic equilibrium*, pages 308–329, 1992.

[DS10] Ulrich Doraszelski and Mark Satterthwaite. Computable markov-perfect industry dynamics. *The RAND Journal of Economics*, 41(2):215–243, 2010.

[DLP13] Y E Du, Ehud Lehrer, and A D Y Pauzner. Competitive economy as a ranking device over networks. submitted, 2013.

[Dud02] R M Dudley. *Real Analysis and Probability*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2002.

[EG87] Robert F Engle and Clive W J Granger. Co-integration and Error Correction: Representation, Estimation, and Testing. *Econometrica*, 55(2):251–276, 1987.

[EP95] Richard Ericson and Ariel Pakes. Markov-perfect industry dynamics: a framework for empirical work. *The Review of Economic Studies*, 62(1):53–82, 1995.

[ES13] David Evans and Thomas J Sargent. *History dependent public policies*. Oxford University Press, 2013.

[EH01] G W Evans and S Honkapohja. *Learning and Expectations in Macroeconomics*. Frontiers of Economic Research. Princeton University Press, 2001.

[FSTD15] Pablo Fajgelbaum, Edouard Schaal, and Mathieu Taschereau-Dumouchel. Uncertainty traps. Technical Report, National Bureau of Economic Research, 2015.

[Fri56] M. Friedman. *A Theory of the Consumption Function*. Princeton University Press, 1956.

[FF98] Milton Friedman and Rose D Friedman. *Two Lucky People*. University of Chicago Press, 1998.

[Gal37] Albert Gallatin. Report on the finances**, november, 1807. In *Reports of the Secretary of the Treasury of the United States, Vol 1*. Government printing office, Washington, DC, 1837.

[GW10] Marc P Giannoni and Michael Woodford. Optimal target criteria for stabilization policy. Technical Report, National Bureau of Economic Research, 2010.

[Hal78] Robert E Hall. Stochastic Implications of the Life Cycle-Permanent Income Hypothesis: Theory and Evidence. *Journal of Political Economy*, 86(6):971–987, 1978.

[HM82] Robert E Hall and Frederic S Mishkin. The Sensitivity of Consumption to Transitory Income: Estimates from Panel Data on Households. *National Bureau of Economic Research Working Paper Series*, 1982.

[Ham05] James D Hamilton. What's real about the business cycle? *Federal Reserve Bank of St. Louis Review*, pages 435–452, 2005.

[HR85] Dennis Epple, Hansen, Lars. P. and Will Roberds. Linear-quadratic duopoly models of resource depletion. In *Energy, Foresight, and Strategy*. Resources for the Future, vol 1 edition, 1985.

[HS08] L P Hansen and T J Sargent. *Robustness*. Princeton University Press, 2008.

[HS13] L P Hansen and T J Sargent. *Recursive Models of Dynamic Linear Economies*. The Gorman Lectures in Economics. Princeton University Press, 2013.

[Han07] Lars Peter Hansen. Beliefs, Doubts and Learning: Valuing Macroeconomic Risk. *American Economic Review*, 97(2):1–30, May 2007. URL: https://ideas.repec.org/a/aea/aecrev/v97y2007i2p1-30.html, doi:.

[HHL08] Lars Peter Hansen, John C. Heaton, and Nan Li. Consumption Strikes Back? Measuring Long-Run Risk. *Journal of Political Economy*, 116(2):260–302, 04 2008. URL: https://ideas.repec.org/a/ucp/jpolec/v116y2008i2p260-302.html, doi:.

[HR87] Lars Peter Hansen and Scott F Richard. The Role of Conditioning Information in Deducing Testable. *Econometrica*, 55(3):587–613, May 1987.

[HS80] Lars Peter Hansen and Thomas J Sargent. Formulating and estimating dynamic linear rational expectations models. *Journal of Economic Dynamics and control*, 2:7–46, 1980.

[HS00] Lars Peter Hansen and Thomas J Sargent. Wanting robustness in macroeconomics. *Manuscript, Department of Economics, Stanford University.*, 2000.

[HS17] Lars Peter Hansen and Thomas J. Sargent. *Risk, Uncertainty, and Value*. Princeton University Press, Princeton, New Jersey, 2017.

[HS09] Lars Peter Hansen and Jose A. Scheinkman. Long-term risk: an operator approach. *Econometrica*, 77(1):177–234, 01 2009.

[HK78] J. Michael Harrison and David M. Kreps. Speculative investor behavior in a stock market with heterogeneous expectations. *The Quarterly Journal of Economics*, 92(2):323–336, 1978.

[HK79] J. Michael Harrison and David M. Kreps. Martingales and arbitrage in multiperiod securities markets. *Journal of Economic Theory*, 20(3):381–408, June 1979.

[HL96] John Heaton and Deborah J Lucas. Evaluating the effects of incomplete markets on risk sharing and asset pricing. *Journal of Political Economy*, pages 443–487, 1996.

[HK85] Elhanan Helpman and Paul Krugman. *Market structure and international trade*. MIT Press Cambridge, 1985.

[HLL96] O Hernandez-Lerma and J B Lasserre. *Discrete-Time Markov Control Processes: Basic Optimality Criteria*. Number Vol 1 in Applications of Mathematics Stochastic Modelling and Applied Probability. Springer, 1996.

[HP92] Hugo A Hopenhayn and Edward C Prescott. Stochastic Monotonicity and Stationary Distributions for Dynamic Economies. *Econometrica*, 60(6):1387–1406, 1992.

[HR93] Hugo A Hopenhayn and Richard Rogerson. Job Turnover and Policy Evaluation: A General Equilibrium Analysis. *Journal of Political Economy*, 101(5):915–938, 1993.

[Hug93] Mark Huggett. The risk-free rate in heterogeneous-agent incomplete-insurance economies. *Journal of Economic Dynamics and Control*, 17(5-6):953–969, 1993.

[Haggstrom02] Olle Häggström. *Finite Markov chains and algorithmic applications*. Volume 52. Cambridge University Press, 2002.

[JYC88] Robert J. Shiller John Y. Campbell. The Dividend-Price Ratio and Expectations of Future Dividends and Discount Factors. *Review of Financial Studies*, 1(3):195–228, 1988.

[Jov79] Boyan Jovanovic. Firm-specific capital and turnover. *Journal of Political Economy*, 87(6):1246–1260, 1979.

[Jr03] Robert E. Lucas Jr. Macroeconomic Priorities. *American Economic Review*, 93(1):1–14, March 2003. URL: https://ideas.repec.org/a/aea/aecrev/v93y2003i1p1-14.html, doi:.

[Jud90] K L Judd. Cournot versus bertrand: a dynamic resolution. Technical Report, Hoover Institution, Stanford University, 1990.

[Jud85] Kenneth L Judd. On the performance of patents. *Econometrica*, pages 567–585, 1985.

[Janich94] K Jänich. *Linear Algebra*. Springer Undergraduate Texts in Mathematics and Technology. Springer, 1994.

[Kam12] Takashi Kamihigashi. Elementary results on solutions to the bellman equation of dynamic programming: existence, uniqueness, and convergence. Technical Report, Kobe University, 2012.

[Kre88] David M. Kreps. *Notes on the Theory of Choice*. Westview Press, Boulder, Colorado, 1988.

[Kuh13] Moritz Kuhn. Recursive Equilibria In An Aiyagari-Style Economy With Permanent Income Shocks. *International Economic Review*, 54:807–835, 2013.

[KP80a] Finn E Kydland and Edward C Prescott. Dynamic optimal taxation, rational expectations and optimal control. *Journal of Economic Dynamics and Control*, 2:79–91, 1980.

[KP77] Finn E., Kydland and Edward C. Prescott. Rules rather than discretion: the inconsistency of optimal plans. *Journal of Political Economy*, 106(5):867–896, 1977.

[KP80b] Finn E., Kydland and Edward C. Prescott. Time to build and aggregate fluctuations. *Econometrics*, 50(6):1345–2370, 1980.

[LM94] A Lasota and M C MacKey. *Chaos, Fractals, and Noise: Stochastic Aspects of Dynamics*. Applied Mathematical Sciences. Springer-Verlag, 1994.

[LL01] Martin Lettau and Sydney Ludvigson. Consumption, Aggregate Wealth, and Expected Stock Returns. *Journal of Finance*, 56(3):815–849, 06 2001.

[LL04] Martin Lettau and Sydney C. Ludvigson. Understanding Trend and Cycle in Asset Values: Reevaluating the Wealth Effect on Consumption. *American Economic Review*, 94(1):276–299, March 2004.

[LM80] David Levhari and Leonard J Mirman. The great fish war: an example using a dynamic cournot-nash solution. *The Bell Journal of Economics*, pages 322–334, 1980.

[LS12] L Ljungqvist and T J Sargent. *Recursive Macroeconomic Theory*. MIT Press, 3 edition, 2012.

[Luc78] Robert E Lucas, Jr. Asset prices in an exchange economy. *Econometrica: Journal of the Econometric Society*, 46(6):1429–1445, 1978.

[LP71] Robert E Lucas, Jr and Edward C Prescott. Investment under uncertainty. *Econometrica: Journal of the Econometric Society*, pages 659–681, 1971.

[LS83] Robert E Lucas, Jr and Nancy L Stokey. Optimal Fiscal and Monetary Policy in an Economy without Capital. *Journal of monetary Economics*, 12(3):55–93, 1983.

[MS89] Albert Marcet and Thomas J Sargent. Convergence of Least-Squares Learning in Environments with Hidden State Variables and Private Information. *Journal of Political Economy*, 97(6):1306–1322, 1989.

[MdRV10] V Filipe Martins-da-Rocha and Yiannis Vailakis. Existence and Uniqueness of a Fixed Point for Local Contractions. *Econometrica*, 78(3):1127–1141, 2010.

[MCWG95] A Mas-Colell, M D Whinston, and J R Green. *Microeconomic Theory*. Volume 1. Oxford University Press, 1995.

[McC70] J J McCall. Economics of Information and Job Search. *The Quarterly Journal of Economics*, 84(1):113–126, 1970.

[MT09] S P Meyn and R L Tweedie. *Markov Chains and Stochastic Stability*. Cambridge University Press, 2009.

[MS85] Marcus Miller and Mark Salmon. Dynamic Games and the Time Inconsistency of Optimal Policy in Open Economies. *Economic Journal*, 95:124–137, 1985.

[MF02] Mario J Miranda and P L Fackler. *Applied Computational Economics and Finance*. Cambridge: MIT Press, 2002.

[MB54] F. Modigliani and R. Brumberg. Utility analysis and the consumption function: An interpretation of cross-section data. In K.K Kurihara, editor, *Post-Keynesian Economics*. 1954.

[Mut60] John F Muth. Optimal properties of exponentially weighted forecasts. *Journal of the american statistical association*, 55(290):299–306, 1960.

[Nea99] Derek Neal. The Complexity of Job Mobility among Young Men. *Journal of Labor Economics*, 17(2):237–261, 1999.

[Orf88] Sophocles J Orfanidis. *Optimum Signal Processing: An Introduction*. McGraw Hill Publishing, New York, New York, 1988.

[Par99] Jonathan A Parker. The Reaction of Household Consumption to Predictable Changes in Social Security Taxes. *American Economic Review*, 89(4):959–973, 1999.

[PL92] D.A. Currie, Pearlman, J.G. and P.L. Levine. Rational expectations with partial information. *Economic Modeling*, 3:90–105, 1992.

[Pea92] J.G. Pearlman. Reputational and nonreputational policies under partial information. *Journal of Economic Dynamics and Control*, 16(2):339–358, 1992.

[Pre77] Edward C. Prescott. Should control theory be used for economic stabilization? *Journal of Monetary Economics*, 7:13–38, 1977.

[Put05] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2005.

[PalS13] Jenő Pál and John Stachurski. Fitted value function iteration with probability one contractions. *Journal of Economic Dynamics and Control*, 37(1):251–264, 2013.

[Rab02] Guillaume Rabault. When do borrowing constraints bind? Some new results on the income fluctuation problem. *Journal of Economic Dynamics and Control*, 26(2):217–245, 2002.

[Ram27] F. P. Ramsey. A Contribution to the theory of taxation. *Economic Journal*, 37(145):47–61, 1927.

[Rei09] Michael Reiter. Solving heterogeneous-agent models by projection and perturbation. *Journal of Economic Dynamics and Control*, 33(3):649–665, 2009.

[Rom05] Steven Roman. *Advanced linear algebra*. Volume 3. Springer, 2005.

[Roz67] Y. A. Rozanov. *Stationary Random Processes*. Holden-Day, San Francisco, 1967.

[Rus96] John Rust. Numerical dynamic programming in economics. *Handbook of computational economics*, 1:619–729, 1996.

[Rya12] Stephen P Ryan. The costs of environmental regulation in a concentrated industry. *Econometrica*, 80(3):1019–1061, 2012.

[Sar79] T J Sargent. A note on maximum likelihood estimation of the rational expectations model of the term structure. *Journal of Monetary Economics*, 35:245–274, 1979.

[Sar87] T J Sargent. *Macroeconomic Theory*. Academic Press, 2nd edition, 1987.

[SE77] Jack Schechtman and Vera L S Escudero. Some results on an income fluctuation problem. *Journal of Economic Theory*, 16(2):151–166, 1977.

[Sch14] Jose A. Scheinkman. *Speculation, Trading, and Bubbles*. Columbia University Press, New York, 2014.

[Sch69] Thomas C Schelling. Models of Segregation. *American Economic Review*, 59(2):488–493, 1969.

[Shi95] A N Shiriaev. *Probability*. Graduate texts in mathematics. Springer. Springer, 2nd edition, 1995.

[SLP89] N L Stokey, R E Lucas, and E C Prescott. *Recursive Methods in Economic Dynamics*. Harvard University Press, 1989.

[Sto89] Nancy L Stokey. Reputation and time consistency. *The American Economic Review*, pages 134–139, 1989.

[STY04] Kjetil Storesletten, Christopher I Telmer, and Amir Yaron. Consumption and risk sharing over the life cycle. *Journal of Monetary Economics*, 51(3):609–633, 2004.

[Sun96] R K Sundaram. *A First Course in Optimization Theory*. Cambridge University Press, 1996.

[Tal00] Thomas D Tallarini. Risk-sensitive real business cycles. *Journal of Monetary Economics*, 45(3):507–532, June 2000.

[Tau86] George Tauchen. Finite state markov-chain approximations to univariate and vector autoregressions. *Economics Letters*, 20(2):177–181, 1986.

[Tow83] Robert M. Townsend. Forecasting the forecasts of others. *Journal of Political Economy*, 91:546–588, 1983.

[VL11] Ngo Van Long. Dynamic games in the economics of natural resources: a survey. *Dynamic Games and Applications*, 1(1):115–148, 2011.

[Wal47] Abraham Wald. *Sequential Analysis*. John Wiley and Sons, New York, 1947.

[Whi63] Peter Whittle. *Prediction and regulation by linear least-square methods*. English Univ. Press, 1963.

[Whi83] Peter Whittle. *Regulation and Prediction by Linear Least Squares Methods, 2nd edition*. University of Minnesota Press, 1983.

[Woo03] Michael Woodford. *Interest and Prices: Foundations of a Theory of Monetary Policy*. Princeton University Press, 2003.

[Woo15] Jeffrey M Wooldridge. *Introductory econometrics: A modern approach*. Nelson Education, 2015.

[YS05] G Alastair Young and Richard L Smith. *Essentials of statistical inference*. Cambridge University Press, 2005.

INDEX

A

A Problem that Stumped Milton Friedman, 655
A Stochastic Optimal Growth Model, 592, 593
An Introduction to Job Search, 579
AR, 442
ARMA, 440, 442
ARMA Processes, 436

B

Bellman Equation, 725
Bisection, 113

C

Central Limit Theorem, 323, 331
    Intuition, 331
    Multivariate Case, 336
cloud computing, 12, 202
    amazon ec2, 13
    google app engine, 13
    pythonanywhere, 13
    sagemath cloud, 13
    wakari, 12
CLT, 323
Complex Numbers, 441
Consumption
    Tax, 880
Continuous State Markov Chains, 368
Covariance Stationary, 437
Covariance Stationary Processes, 435
    AR, 439
    MA, 439
Cython, 187, 196

D

Data Sources, 216
Debugging, 175, 176
Discrete State Dynamic Programming, 547
Dynamic Programming
    Computation, 598
    Shortest Paths, 571
    Theory, 597
    Unbounded Utility, 598
    Value Function Iteration, 598
Dynamic Typing, 188

E

Eigenvalues, 263, 277
Eigenvectors, 263, 277
Ergodicity, 300, 314

F

Finite Markov Asset Pricing
    Lucas Tree, 837
Finite Markov Chains, 300, 301
    Stochastic Matrices, 301
Fixed Point Theory, 854

G

General Linear Processes, 438
Git, 36

H

History Dependent Public Policies, 939
    Competitive Equilibrium, 942
    Ramsey Timing, 941
    Sequence of Governments Timing, 941
    Timing Protocols, 941

I

Immutable, 155
Integration, 108, 116
Introduction to Asset Pricing, 830
IPython, 16
    Shell, 33
Irreducibility and Aperiodicity, 300, 308

J

Jupyter, 14, 16
Jupyter Notebook
    Basics, 20
    Help, 25
    nbviewer, 27
    Setup, 18
    Sharing, 27

K

Kalman Filter, 389
    Programming Implementation, 396
    Recursive Procedure, 395

L

Lake Model, 756
Law of Large Numbers, 323, 324
    Illustration, 326
    Multivariate Case, 336
    Proof, 325
Linear Algebra, 108, 117, 263
    Differentiating Linear and Quadratic Forms, 280
    Eigenvalues, 277
    Eigenvectors, 277
    Matrices, 270
    Matrix Norms, 279
    Neumann's Theorem, 279
    Positive Definite Matrices, 280
    SciPy, 276
    Series Expansions, 279
    Spectral Radius, 279
    Vectors, 264
Linear Markov Perfect Equilibria, 814
Linear State Space Models, 342
    Distributions, 348, 349
    Ergodicity, 354
    Martingale Difference Shocks, 344
    Moments, 348
    Moving Average Representations, 348
    Prediction, 359
    Seasonals, 346
    Stationarity, 354
    Time Trends, 347
    Univariate Autoregressive Processes, 345
    Vector Autoregressions, 346
LLN, 323
LQ Control, 405
    Infinite Horizon, 416
    Optimality (Finite Horizon), 409
Lucas Model, 850
    Assets, 850
    Computation, 854
    Consumers, 850
    Dynamic Program, 851
    Equilibrium Constraints, 852
    Equilibrium Price Funtion, 852
    Pricing, 851
    Solving, 853

M

MA, 442
Marginal Distributions, 300, 306
Markov Asset Pricing
    Overview, 830
Markov Chains, 301
    Calculating Stationary Distributions, 312
    Continuous State, 368
    Convergence to Stationarity, 313
    Cross-Sectional Distributions, 307
    Ergodicity, 314
    Forecasting Future Values, 314
    Future Probabilities, 307
    Irreducibility, Aperiodicity, 308
    Marginal Distributions, 306
    Simulation, 303
    Stationary Distributions, 311
Markov Perfect Equilibrium, 812
    Applications, 816
    Background, 812
    Overview, 812
Matplotlib, 6, 95
    3D Plots, 104
    Multiple Plots on One Axis, 102
    Object-Oriented API, 98
    Simple API, 96
    Subplots, 103
Matrix
    Determinants, 275
    Inverse, 274
    Maps, 272
    Numpy, 271
    Operations, 270
    Solving Systems of Equations, 273
McCall Model, 709
Modeling

Career Choice, 911
Models
Additive functionals, 464
Code style, 180
Harrison Kreps, 862
Linear State Space, 343
Lucas Asset Pricing, 849
Markov Asset Pricing, 830
McCall, 709
Multiplicative functionals, 482
On-the-Job Search, 898
Permanent Income, 674, 689
Pricing, 830
Schelling's Segregation Model, 745
Sequential analysis, 655
Mutable, 155

N

NetworkX, 11
Neumann's Theorem, 279
Newton-Raphson Method, 114
Nonparametric Estimation, 454
Numba, 187, 193
NumPy, 75, 76, 107, 108
Arrays, 77
Arrays (Creating), 79
Arrays (Indexing), 80
Arrays (Methods), 82
Arrays (Operations), 84
Arrays (Shape and Dimension), 78
Comparisons, 89
Matrix Multiplication, 85
Universal Functions, 191
Vectorized Functions, 88

O

Object Oriented Programming, 119
Classes, 122
Key Concepts, 120
Methods, 126
Special Methods, 134
Terminology, 120
On-the-Job Search, 898
Model, 899
Model Features, 899
Parameterization, 899
Programming Implementation, 900
Solving for Policies, 905
Optimal Growth
Model, 593
Policy Function, 608
Policy Funtion Approach, 595
Optimal Savings
Computation, 640
Problem, 638
Programming Implementation, 642
Optimal Taxation in an LQ Economy, 992
Optimization, 108, 115
Multivariate, 116
Orthogonal Projection, 283

P

Pandas, 10, 207
Accessing Data, 219
DataFrames, 210
Series, 209
Pandas for Panel Data, 223
parallel computing, 13, 202
copperhead, 13
ipython, 13
pycuda, 13
starcluster, 13
Periodograms, 450, 451
Computation, 452
Interpretation, 451
Permanent Income Model
Hall's Representation, 682
Savings Problem, 675
Permanent Income Model: I, 674
Permanent Income Model: II, 689
Positive Definite Matrices, 280
Pricing Models, 830
Risk Aversion, 831
Risk Neutral, 831
pyMC, 11
pystan, 11
Python, 14
Anaconda, 15
Assertions, 157
common uses, 2
Comparison, 65
Conditions, 46
Content, 140
Cython, 196
Data Types, 56
Decorators, 161, 163, 166

Descriptors, 161, 164
Dictionaries, 60
Docstrings, 68
Exceptions, 158
For loop, 43
Functions, 45, 67
Generator Functions, 168
Generators, 167
Handling Errors, 157
Identity, 141
import, 60
Indentation, 43
Interfacing with Fortran, 202
Interpreter, 151
Introductory Example, 37
IO, 61
IPython, 16
Iterables, 145
Iteration, 63, 143
Iterators, 143, 144, 147
keyword arguments, 70
lambda functions, 69
List comprehension, 47
Lists, 41
Logical Expressions, 66
Matplotlib, 95
Methods, 141
Namespace (__builtins__), 153
Namespace (Global), 152
Namespace (Local), 152
Namespace (Resolution), 153
Namespaces, 149
Nuitka, 205
Numba, 193
NumPy, 75
Object Oriented Programming, 119
Objects, 138
Packages, 39
Pandas, 207, 223
Paths, 62
PEP8, 71
Properties, 166
PyPI, 13
PyPy, 205
Pyston, 205
Pythran, 205
Recursion, 172
requests, 216
Runtime Errors, 158
SciPy, 90, 107
Sets, 60
Slicing, 59
Subpackages, 40
syntax and design, 5
Tuples, 58
Type, 139
Variable Names, 147
Vectorization, 190
While loop, 44
python, 1

Q

QuantEcon, 28
Installation, 28

R

Ramsey Problem, 940, 943
Computing, 944
Credible Policy, 955
Optimal Taxation, 992
Recursive Representation, 947
Time Inconsistency, 952
Two Subproblems, 945
Rational Expectations Equilibrium, 798
Competitive Equilbrium (w. Adjustment Costs), 801
Computation, 804
Definition, 800
Planning Problem Approach, 804
requests, 216
Robustness, 724

S

Schelling Segregation Model, 745
scientific programming, 5
Blaze, 13
CVXPY, 13
Jupyter, 13
Numba, 13
numeric, 5
PyTables, 13
scikit-learn, 11
SciPy, 90, 107, 108
Bisection, 113
Fixed Points, 115
Integration, 116

Linear Algebra, 117
Multivariate Root Finding, 115
Newton-Raphson Method, 114
Optimization, 115
Statistics, 109
Smoothing, 450, 454
Spectra, 450
Estimation, 450
Spectra, Estimation
AR(1) Setting, 458
Fast Fourier Transform, 450
Pre-Filtering, 458
Smoothing, 454, 455, 458
Spectral Analysis, 435, 436, 441
Spectral Densities, 441
Spectral Density, 443
interpretation, 443
Inverting the Transformation, 445
Mathematical Theory, 446
Spectral Radius, 279
Static Types, 188
Stationary Distributions, 300, 311
statsmodels, 11
Stochastic Matrices, 301
SymPy, 9

T

Text Editors, 33
The Endogenous Grid Method, 628
The Income Fluctuation Problem, 637
Time Iteration with Euler Equations, 611

U

Unbounded Utility, 598

V

Vectorization, 187, 190
Operations on Arrays, 190
Vectors, 263, 264
Inner Product, 266
Linear Independence, 269
Norm, 266
Operations, 265
Span, 267

W

White Noise, 437, 442
Wold's Decomposition, 438