

# QUANTITATIVE ECONOMICS with Python

Thomas Sargent and John Stachurski

July 20, 2016



## CONTENTS

<b>1 Programming in Python</b>	<b>7</b>
1.1 About Python . . . . .	7
1.2 Setting up Your Python Environment . . . . .	16
1.3 An Introductory Example . . . . .	35
1.4 Python Essentials . . . . .	48
1.5 Object Oriented Programming . . . . .	63
1.6 How it Works: Data, Variables and Names . . . . .	80
1.7 More Language Features . . . . .	96
1.8 NumPy . . . . .	111
1.9 Matplotlib . . . . .	126
1.10 SciPy . . . . .	137
1.11 Pandas . . . . .	147
1.12 IPython Tips and Tricks . . . . .	161
1.13 The Need for Speed . . . . .	169
<b>2 Introductory Applications</b>	<b>183</b>
2.1 Linear Algebra . . . . .	183
2.2 Finite Markov Chains . . . . .	200
2.3 Orthogonal Projection and its Applications . . . . .	220
2.4 Shortest Paths . . . . .	230
2.5 The McCall Job Search Model . . . . .	234
2.6 Schelling's Segregation Model . . . . .	243
2.7 LLN and CLT . . . . .	248
2.8 Linear State Space Models . . . . .	261
2.9 A Lake Model of Employment and Unemployment . . . . .	287
2.10 A First Look at the Kalman Filter . . . . .	303
2.11 Uncertainty Traps . . . . .	318
2.12 A Simple Optimal Growth Model . . . . .	325
2.13 Optimal Growth Part II: Adding Some Bling . . . . .	336
2.14 LQ Dynamic Programming Problems . . . . .	346
2.15 Discrete Dynamic Programming . . . . .	373
2.16 Rational Expectations Equilibrium . . . . .	385
2.17 Markov Perfect Equilibrium . . . . .	394
2.18 Markov Asset Pricing . . . . .	407
2.19 A Harrison-Kreps (1978) Model of Asset Prices . . . . .	420

2.20 The Permanent Income Model . . . . .	428
<b>3 Advanced Applications</b>	<b>443</b>
3.1 Continuous State Markov Chains . . . . .	443
3.2 The Lucas Asset Pricing Model . . . . .	458
3.3 The Aiyagari Model . . . . .	468
3.4 Modeling Career Choice . . . . .	477
3.5 On-the-Job Search . . . . .	486
3.6 Search with Offer Distribution Unknown . . . . .	496
3.7 Optimal Savings . . . . .	507
3.8 Covariance Stationary Processes . . . . .	520
3.9 Estimation of Spectra . . . . .	538
3.10 Robustness . . . . .	551
3.11 Dynamic Stackelberg Problems . . . . .	576
3.12 Optimal Taxation . . . . .	591
3.13 History Dependent Public Policies . . . . .	607
3.14 Optimal Taxation with State-Contingent Debt . . . . .	628
3.15 Optimal Taxation without State-Contingent Debt . . . . .	654
3.16 Default Risk and Income Fluctuations . . . . .	670
<b>4 Solutions</b>	<b>685</b>
<b>5 FAQs / Useful Resources</b>	<b>687</b>
5.1 FAQs . . . . .	687
5.2 How do I install Python? . . . . .	687
5.3 How do I start Python? . . . . .	687
5.4 How can I get help on a Python command? . . . . .	687
5.5 Where do I get all the Python programs from the lectures? . . . . .	687
5.6 What's Git? . . . . .	688
5.7 Other Resources . . . . .	688
5.8 IPython Magics . . . . .	688
5.9 IPython Cell Magics . . . . .	688
5.10 Useful Links . . . . .	688
<b>References</b>	<b>691</b>

**Note: You are currently viewing an automatically generated PDF version of our on-line lectures, which are located at**

<http://quant-econ.net>

Please visit the website for more information on the aims and scope of the lectures and the two language options (Julia or Python). This PDF is generated from a set of

source files that are oriented towards the website and to HTML output. As a result, the presentation quality can be less consistent than the website.



---

CHAPTER  
ONE

---

## PROGRAMMING IN 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

When we show you Python code, it is **not** our intention that you seek to follow all the details, or try to replicate all you see

We will work through all of the Python material step by step 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

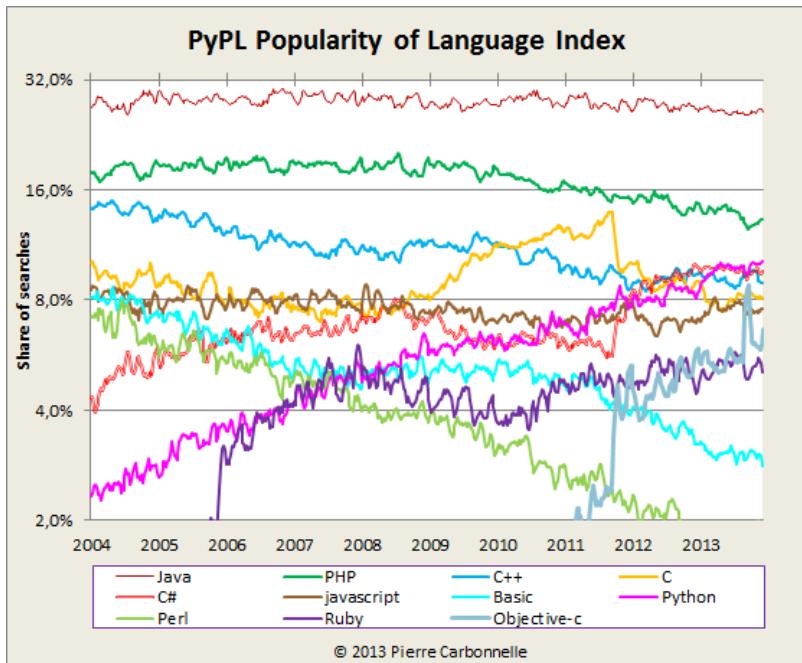
Python is free and open source

Community-based development of the core language is coordinated through the [Python Software Foundation](#)

Python is supported by a vast collection of [standard](#) and [external](#) software libraries

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

This [alternative index](#) gives some indication of the trend



**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.

We'll discuss this more below

## Features

- A high level language suitable for rapid development
- Relatively small core language supported by many libraries
- 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

```
In [1]: import numpy as np # Load the library
In [2]: a = np.linspace(-np.pi, np.pi, 100) # Create array (even grid from -pi to pi)
In [3]: b = np.cos(a) # Apply cosine to each element of a
In [4]: c = np.ones(25) # An array of 25 ones
In [5]: np.dot(c, c) # Compute inner product
Out[5]: 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  $\phi$  is the standard normal density

```
In [5]: from scipy.stats import norm
In [6]: from scipy.integrate import quad
In [7]: phi = norm()
In [8]: value, error = quad(phi.pdf, -2, 2) # Integrate using Gaussian quadrature
In [9]: value
Out[9]: 0.9544997361036417
```

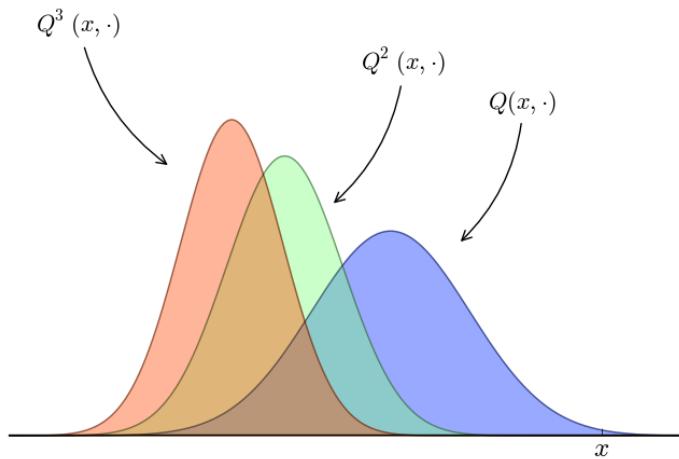
SciPy includes many of the standard routines used in

- 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](#)

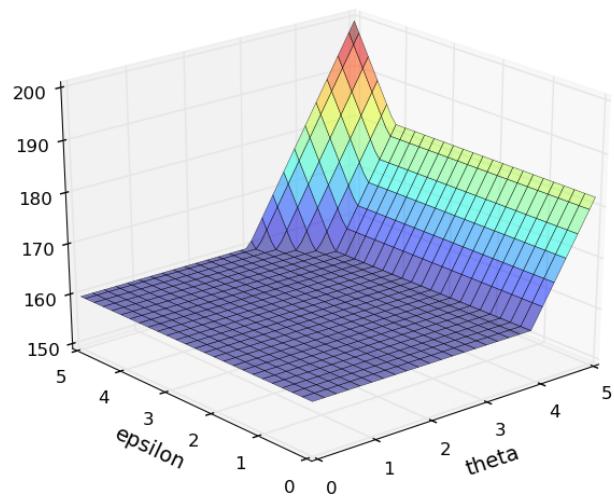
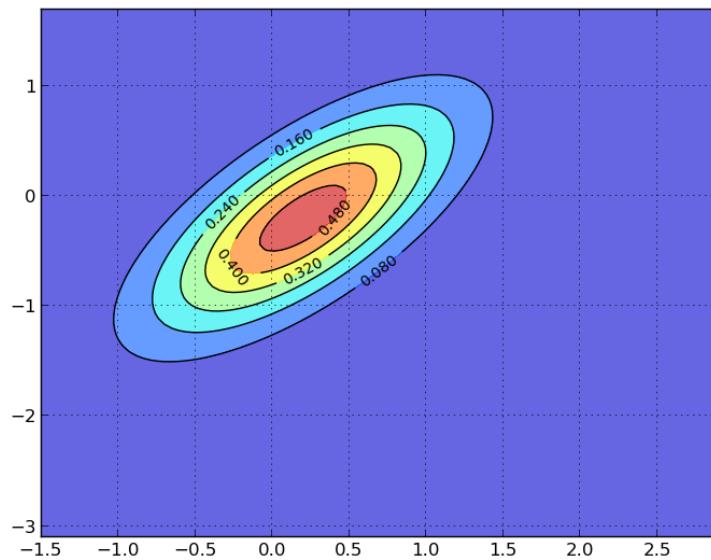
Other graphics libraries include

- 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

```
In [10]: from sympy import Symbol
In [11]: x, y = Symbol('x'), Symbol('y') # Treat 'x' and 'y' as algebraic symbols
In [12]: x + x + x + y
Out[12]: 3*x + y
```



We can manipulate expressions

```
In [13]: expression = (x + y)**2
In [14]: expression.expand()
Out[14]: x**2 + 2*x*y + y**2
```

solve polynomials

```
In [15]: from sympy import solve
In [16]: solve(x**2 + x + 2)
Out[16]: [-1/2 - sqrt(7)*I/2, -1/2 + sqrt(7)*I/2]
```

and calculate limits, derivatives and integrals

```
In [17]: from sympy import limit, sin, diff
In [18]: limit(1 / x, x, 0)
Out[18]: oo
In [19]: limit(sin(x) / x, x, 0)
Out[19]: 1
In [20]: diff(sin(x), x)
Out[20]: 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

```
In [21]: import pandas as pd
In [22]: import scipy as sp
In [23]: data = sp.randn(5, 2) # Create 5x2 matrix of random numbers for toy example
In [24]: dates = pd.date_range('28/12/2010', periods=5)
In [25]: df = pd.DataFrame(data, columns=['price', 'weight'], index=dates)
In [26]: print(df)
          price    weight
2010-12-28  0.45  0.25
2010-12-29  0.55  0.35
2010-12-30  0.65  0.45
2010-12-31  0.75  0.55
2011-01-01  0.85  0.65
```

```
2010-12-28  0.007255  1.129998
2010-12-29 -0.120587 -1.374846
2010-12-30  1.089384  0.612785
2010-12-31  0.257478  0.102297
2011-01-01 -0.350447  1.254644
```

```
In [27]: df.mean()
out[27]:
price      0.176616
weight     0.344975
```

## 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

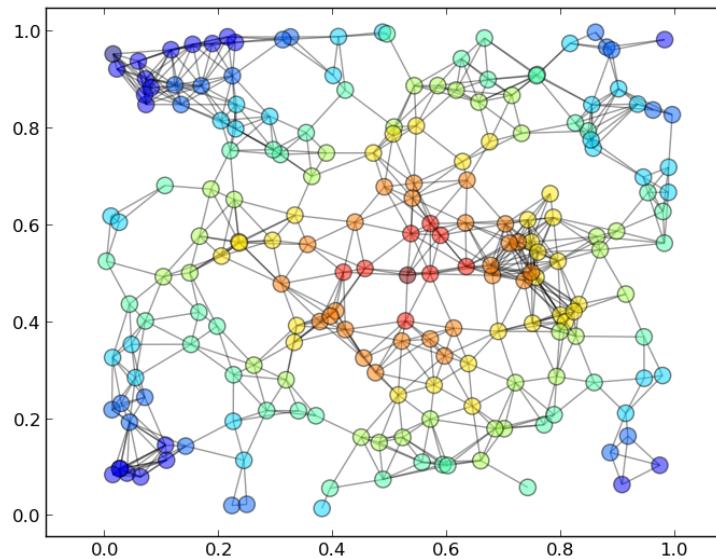
```
"""
Filename: nx_demo.py
Authors: John Stachurski and Thomas J. Sargent
"""

import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

G = nx.random_geometric_graph(200, 0.12) # 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 questions 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. The reason is that these lectures require the entire scientific programming ecosystem, which the core installation doesn't provide.
- To follow all of the code examples in the lectures you need a relatively up to date version of Python and the scientific libraries.

Please read on for instructions on how to get up to date versions of Python and all the major scientific libraries

### First Steps

By far the best approach for our purposes is to install one of the free Python distributions that contains

1. the core Python language **and**
2. the most popular scientific libraries

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, which is currently Python 3.5
- 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

What if you have an older version of Anaconda?

For most scientific programmers, the best thing you can do is [uninstall](#) (see, e.g., [these instructions](#)) and then install the newest version

**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

- Later we'll look at others

Jupyter notebooks provide a *browser-based* interface to Python with

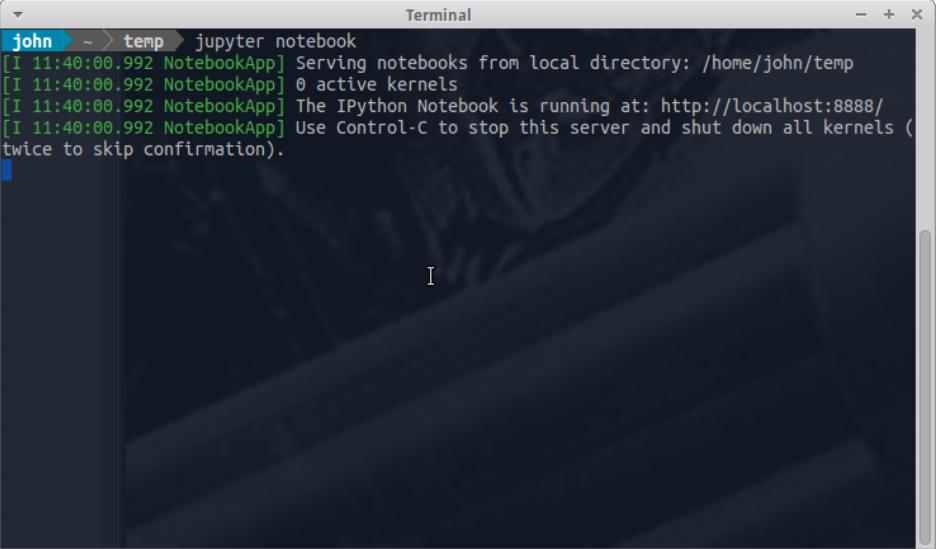
- The ability to write and execute Python commands directly in your browser
- Formatted output also in the browser, including tables, figures, animation, etc.
- The ability to mix in formatted text and mathematical expressions between cells

While Jupyter isn't always the best way to code in Python, it is a great place to start off  
Jupyter is also a powerful tool for organizing and communicating scientific ideas  
In fact Jupyter is fast turning into a major player in scientific computing

- for a slightly over-hyped review, see [this article](#)
- for an example use case in the private sector see Microsoft's [Azure ML Studio](#)
- For more examples of notebooks see the [NB viewer site](#)

**Starting the Jupyter Notebook** To start the Jupyter notebook, open up a terminal (*cmd* for Windows) and type `jupyter notebook`

Here's an example (click to enlarge)



A screenshot of a terminal window titled "Terminal". The window shows the command `jupyter notebook` being run in a directory `temp`. The output indicates that the notebook is serving from the local directory `/home/john/temp`, there are 0 active kernels, and the IPython Notebook is running at `http://localhost:8888/`. It also includes a note about using Control-C to stop the server.

```
john ~ > temp > jupyter notebook
[I 11:40:00.992 NotebookApp] Serving notebooks from local directory: /home/john/temp
[I 11:40:00.992 NotebookApp] 0 active kernels
[I 11:40:00.992 NotebookApp] The IPython Notebook is running at: http://localhost:8888/
[I 11:40:00.992 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
```

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

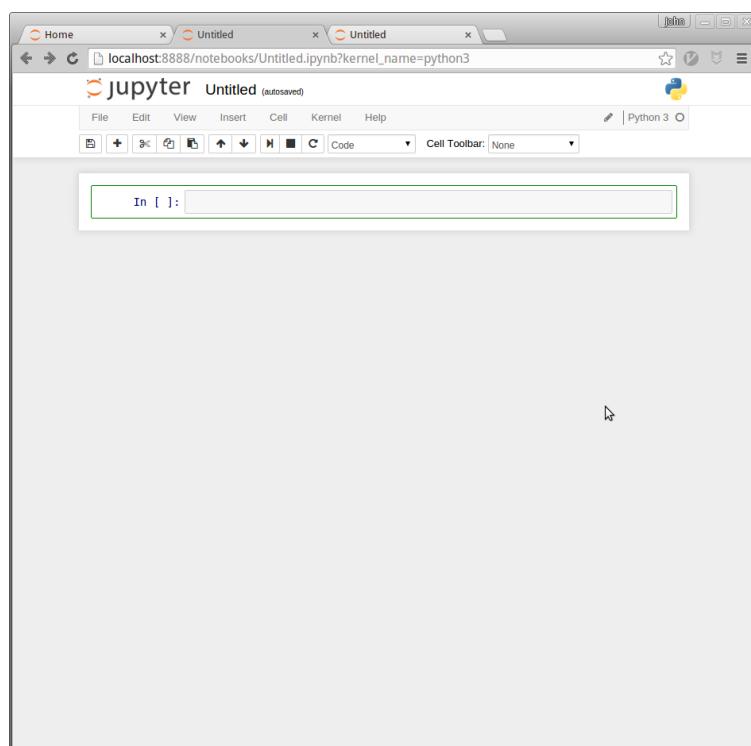
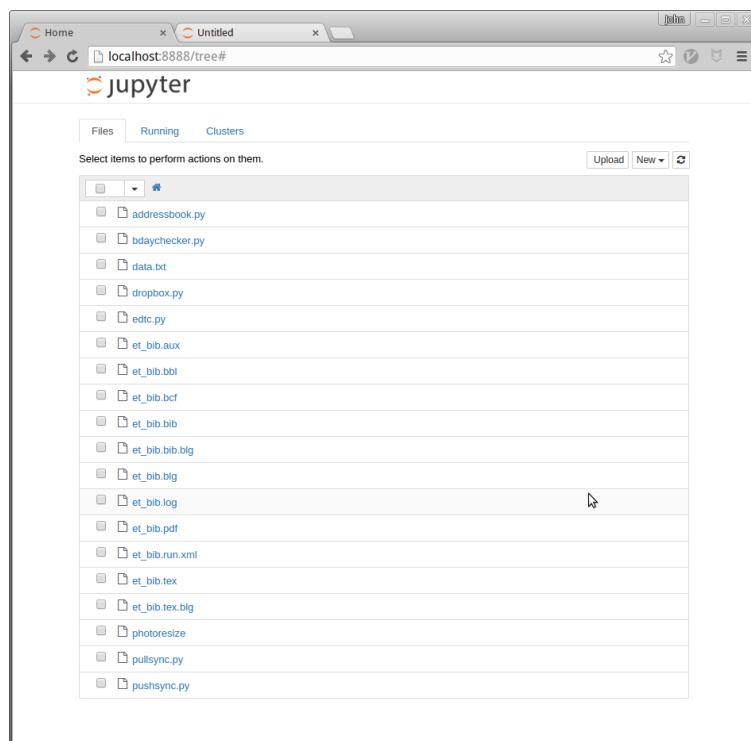
Hopefully your default browser has also opened up with a web page that looks something like this (click to enlarge)

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

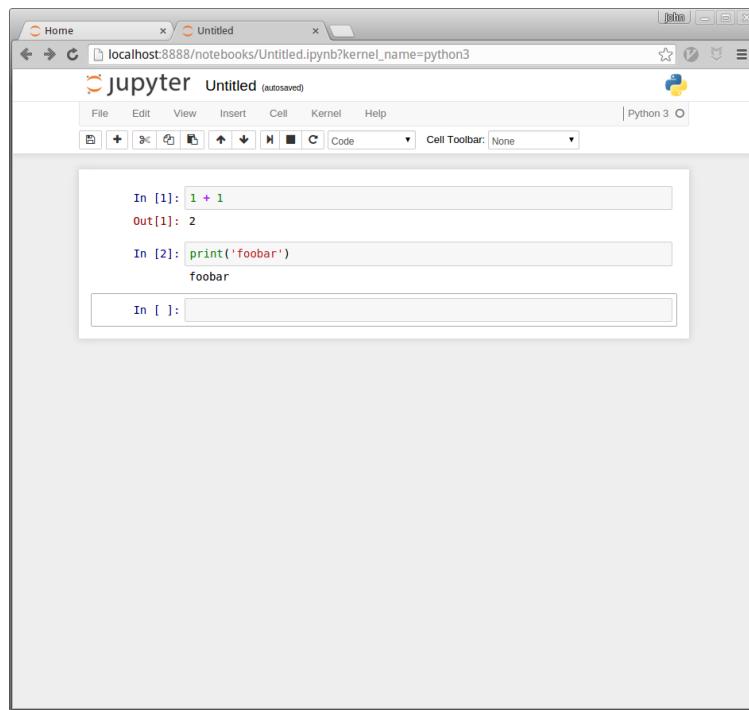
**Notebook Basics** Let's start with how to edit code and run simple programs

**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
- To switch to 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

### 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.py](http://matplotlib.org/1.4.1/examples/pie_and_polar_charts/polar_bar.py)

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)

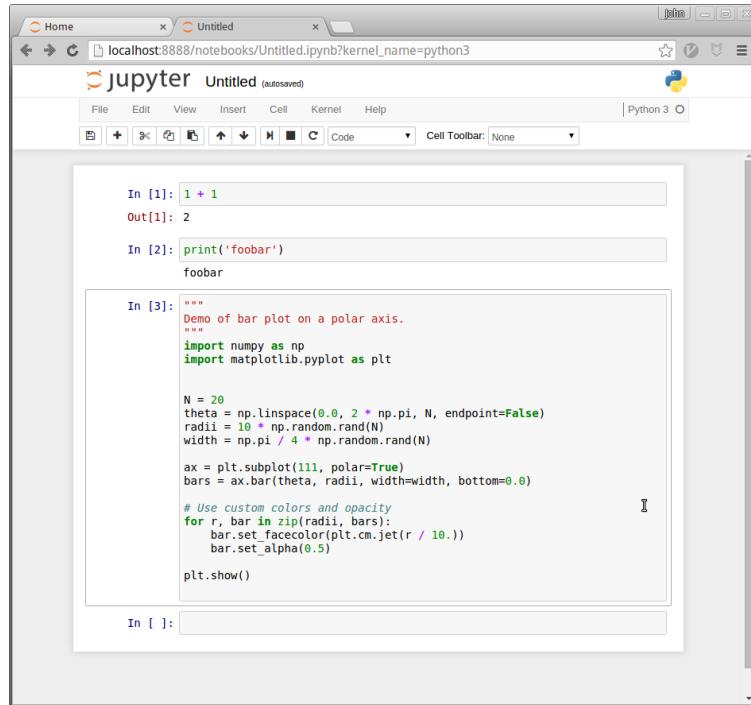
# Use custom colors and opacity
for r, bar in zip(radii, bars):
    bar.set_facecolor(plt.cm.jet(r / 10.))
    bar.set_alpha(0.5)

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, like so

Now Shift-Enter and a figure should appear looking a bit like this



The screenshot shows a Jupyter Notebook window with the following content:

```
In [1]: 1 + 1
Out[1]: 2

In [2]: print('foobar')
foobar

In [3]:
"""
Demo of bar plot on a polar axis.
"""

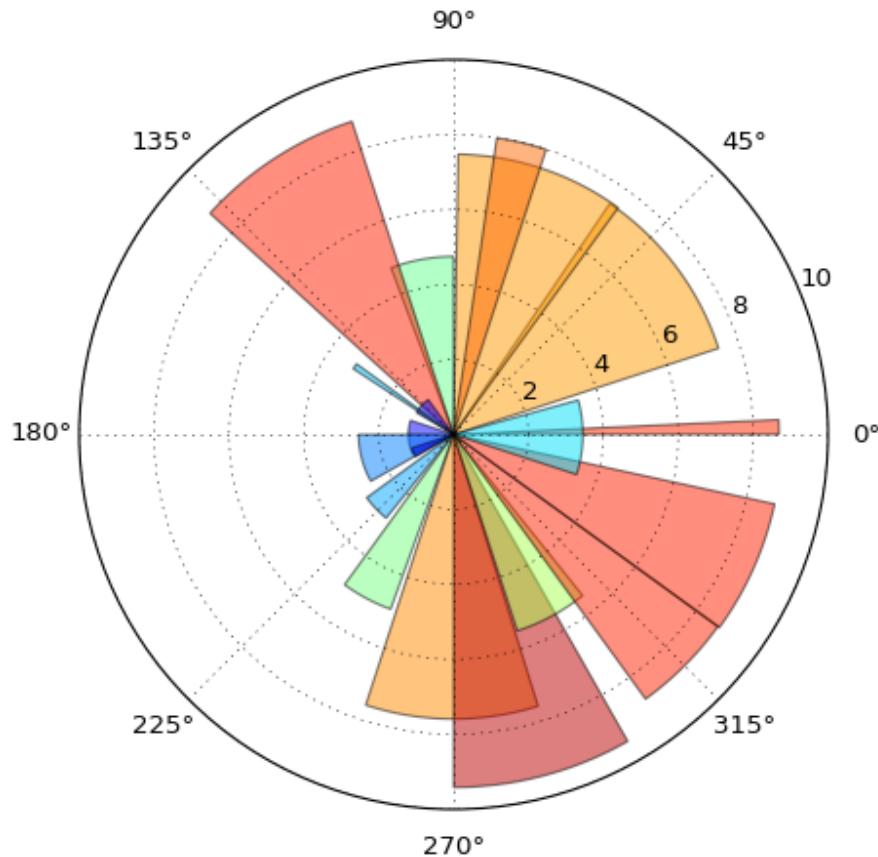
import numpy as np
import matplotlib.pyplot as plt

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()
```



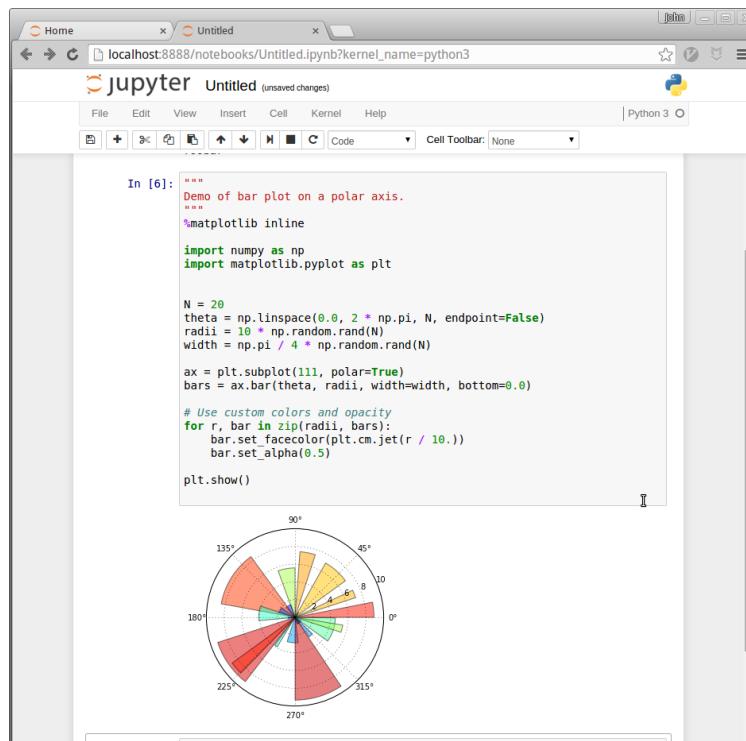
Notes:

- The details of your figure will be different because the data is random
- The figure might be hidden behind your browser — have a look around your desktop

**In-line Figures** One nice thing about Jupyter notebooks is that figures can also be displayed inside the page

To achieve this effect, use the `matplotlib inline` magic

Here we've done this by prepending `%matplotlib inline` to the cell and executing it again (click to enlarge)



**Working with the Notebook** Let's run through a few more notebook essentials

**Tab Completion** One nice feature of Jupyter is tab completion

For example, in the previous program we executed the line `import numpy as np`

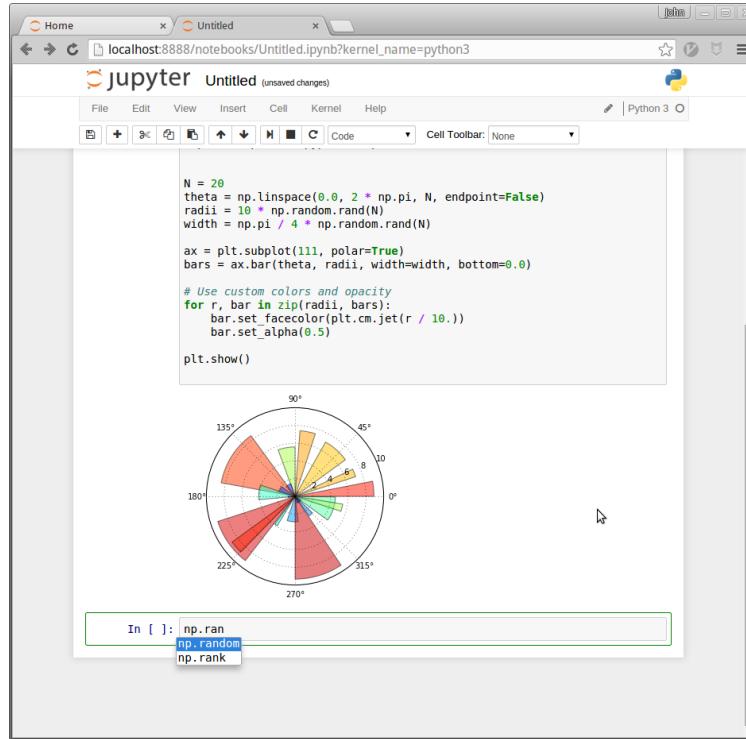
- NumPy is a numerical library we'll work with in depth

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

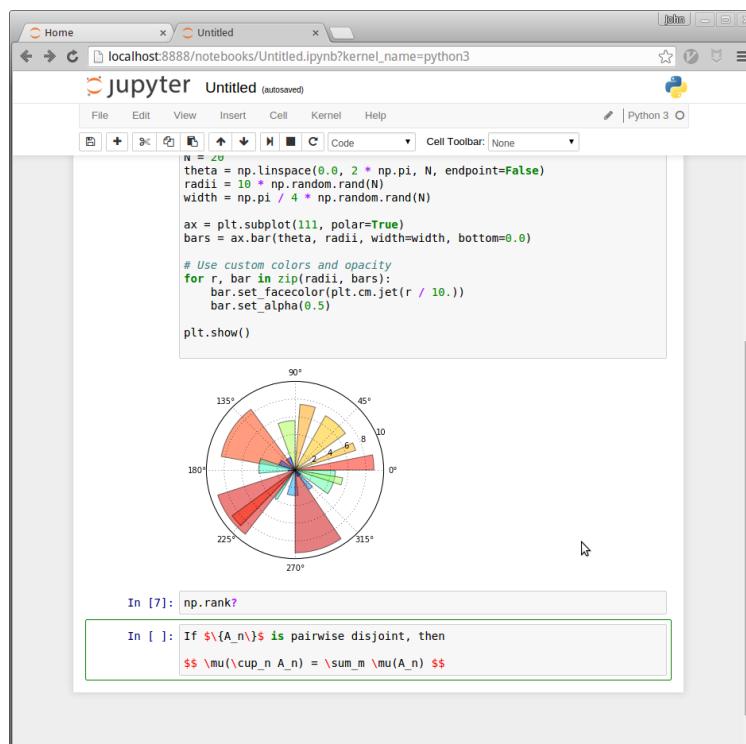
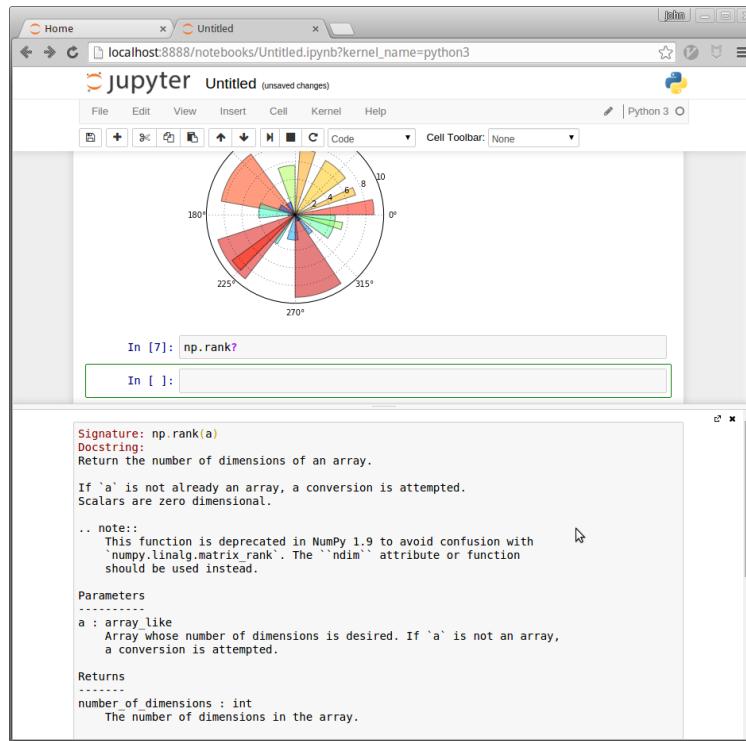
**Other Content** In addition to executing code, the Jupyter notebook allows you to embed text, equations, figures and even videos in the page

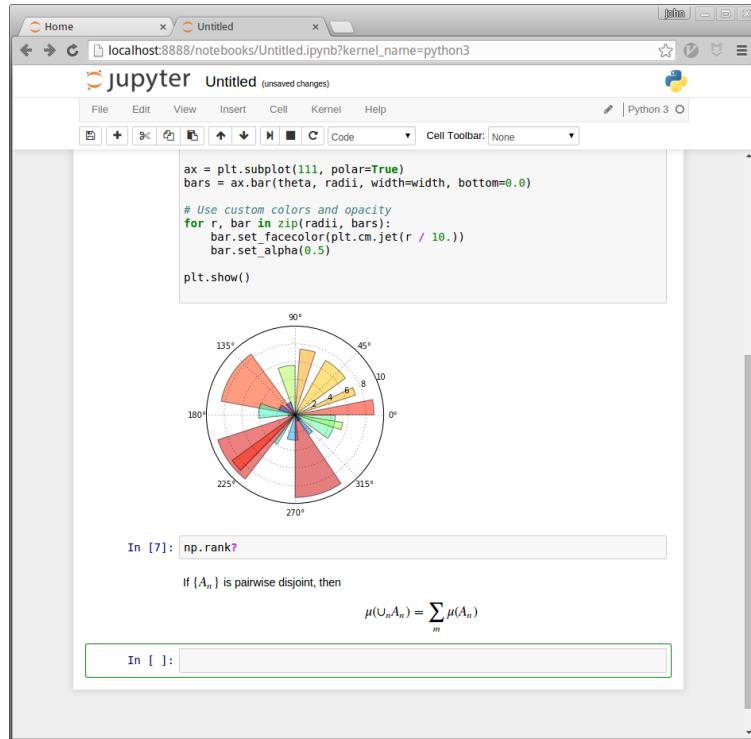
For example, here we enter a mixture of plain text and LaTeX instead of code

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 to know about 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

The code in QuantEcon.py 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](#)
  - The PyPi version of QuantEcon.py is [here](#)

**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](#)

**Other Files** In addition to [QuantEcon.py](#), which contains algorithms, [QuantEcon.applications](#) contains example programs, solutions to exercises and so on

You can download these files individually by navigating to the GitHub page for the individual file

For example, see [here](#) for a file that illustrates eigenvectors

If you like you can then click the Raw button to get a plain text version of the program

However, what you probably want to do is get a copy of the entire repository

**Obtaining the GitHub Repo** One way to do this is to download the zip file by clicking the “Download ZIP” button on the [main page](#)

(Remember where you unzip the directory, and make it somewhere you can find it easily)

There is another, better way to get a copy of the repo, using a program called **Git**

We'll investigate how to do this in *Exercise 2*

**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 usually faster and easier than copy and paste

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

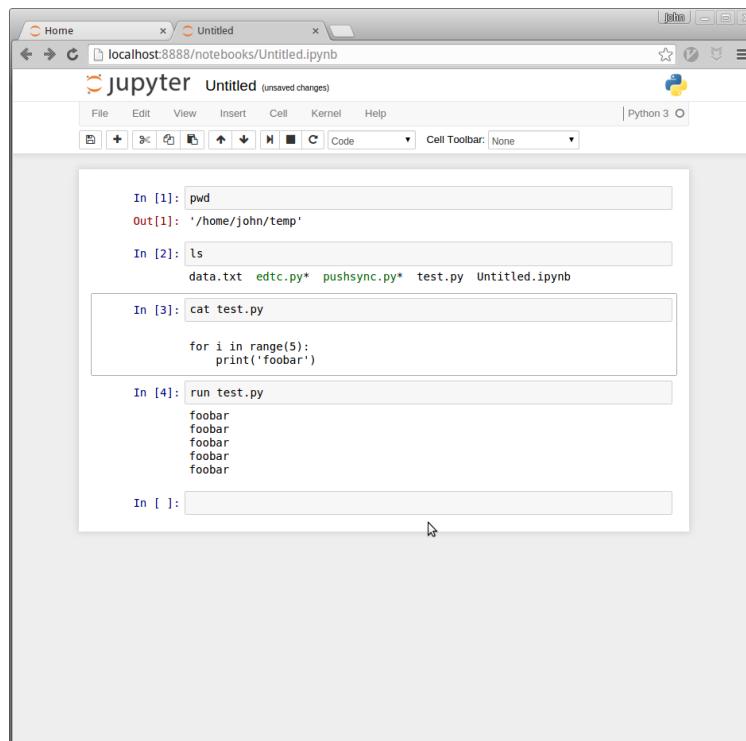
Warnings:

- You might need to replace `run` with `%run`
  - use `%automagic` to toggle the need for `%`
- 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
  - 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
  - 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

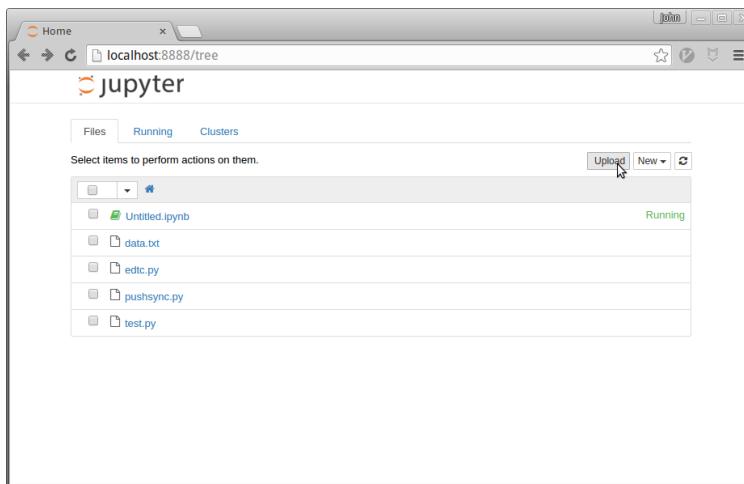
**But file X isn't in my PWD!** If you're trying to run a file not in the present working director, you'll get an error

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`

As an exercise, let's try running the file `white_noise_plot.py`

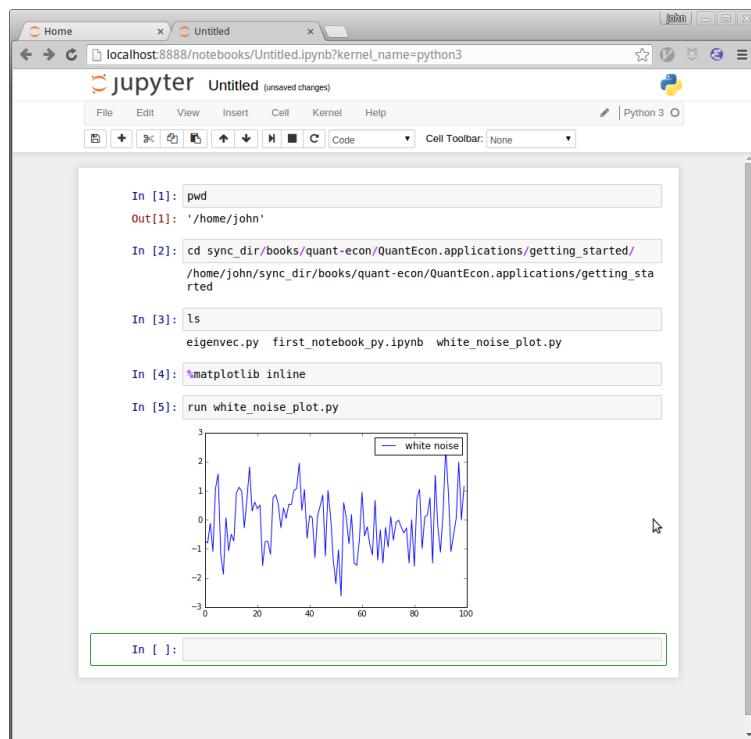
In the figure below, we're working on a Linux machine, and the repo is stored locally in `sync_dir/books/quant-econ/QuantEcon.applications/getting_started`

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

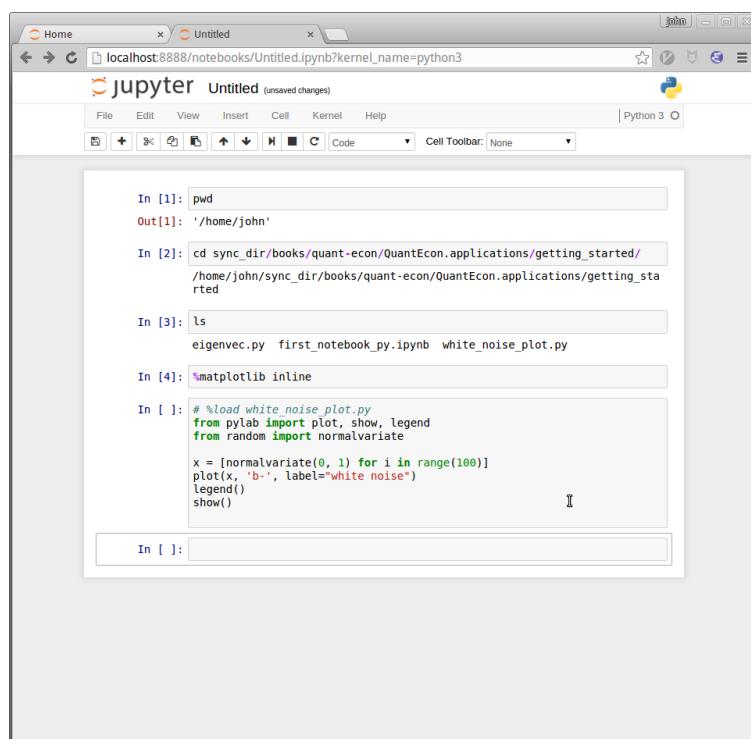
For this purpose we can replace `run white_noise_plot.py` with `load white_noise_plot.py`

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



A screenshot of a Jupyter Notebook interface. The browser title is "jupyter Untitled (unsaved changes)". The URL is "localhost:8888/notebooks/Untitled.ipynb?kernel\_name=python3". The notebook has five cells:

- In [1]: `pwd`  
Out[1]: '/home/john'
- In [2]: `cd sync_dir/books/quant-econ/QuantEcon.applications/getting_started/`  
/home/john/sync\_dir/books/quant-econ/QuantEcon.applications/getting\_stated
- In [3]: `ls`  
eigenvec.py first\_notebook\_py.ipynb white\_noise\_plot.py
- In [4]: `%matplotlib inline`
- In [5]: `run white_noise_plot.py`  
A line plot titled "white noise" showing a blue line with high-frequency oscillations between -3 and 3 on the y-axis and 0 and 100 on the x-axis.



A screenshot of a Jupyter Notebook interface. The browser title is "jupyter Untitled (unsaved changes)". The URL is "localhost:8888/notebooks/Untitled.ipynb?kernel\_name=python3". The notebook has five cells:

- In [1]: `pwd`  
Out[1]: '/home/john'
- In [2]: `cd sync_dir/books/quant-econ/QuantEcon.applications/getting_started/`  
/home/john/sync\_dir/books/quant-econ/QuantEcon.applications/getting\_stated
- In [3]: `ls`  
eigenvec.py first\_notebook\_py.ipynb white\_noise\_plot.py
- In [4]: `%matplotlib inline`
- In [5]: 

```
# %load white_noise_plot.py
from pylab import plot, show, legend
from random import normalvariate

x = [normalvariate(0, 1) for i in range(100)]
plot(x, "b-", label="white noise")
legend()
show()
```

**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 an IPython [cell magic](#)

## Alternatives

The preceding discussion covers most of what you need to know to write and run Python code

However, as you start to write longer programs, you might want to experiment with your workflow

There are many different options and we cover only a few

**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.

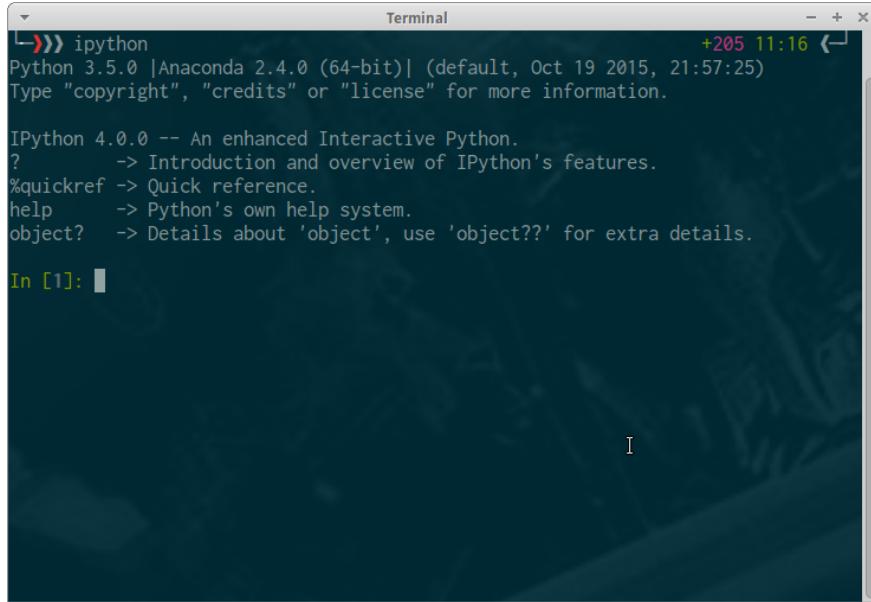
It also has command history through the arrow key

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

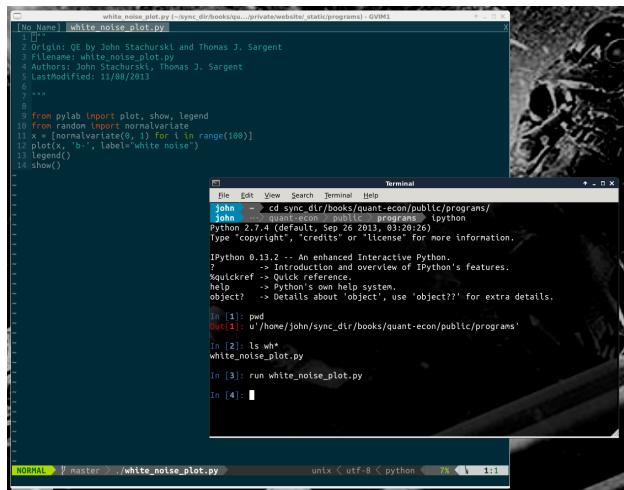
This saves a lot of typing...

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

**Getting the Repo with Git** `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.applications` 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.applications
```

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

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](#)
  - [First Example: Plotting a White Noise Process](#)
  - [Exercises](#)
  - [Solutions](#)

We're now ready to start learning the Python language itself, and the next few lectures are devoted to this task

The level will best suit readers with some basic knowledge of programming concepts

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—how things work—will be covered in later lectures

In reading the following, you should be conscious of the fact that all “first programs” are to some extent contrived

We try to avoid this, but nonetheless

- Be aware that the programs are written to illustrate certain concepts

- Soon you will be writing the same programs in a rather different—and more efficient—way

In particular, the scientific libraries will allow us to accomplish the same things faster and more efficiently, once we know how to use them

However, you also need to learn pure Python, the core language

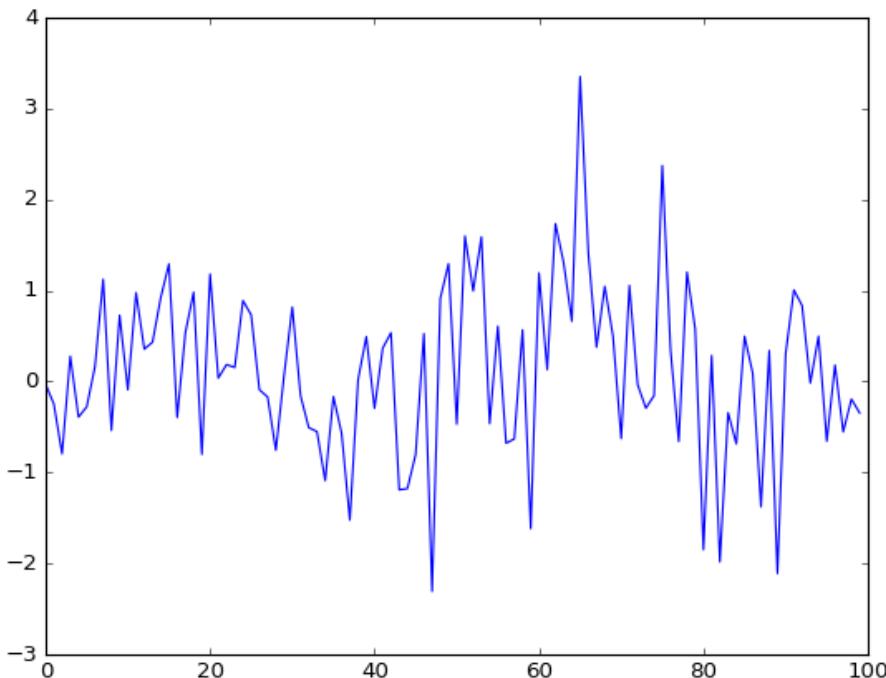
This is the objective of the present lecture, and the next few lectures too

Prerequisites: The [lecture](#) on getting started with Python

### First Example: Plotting a White Noise Process

To begin, suppose we want to simulate and plot the white noise process  $\epsilon_0, \epsilon_1, \dots, \epsilon_T$ , where each draw  $\epsilon_t$  is independent standard normal

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



A program that accomplishes what we want can be found in the file [test\\_program\\_1.py](#) from the [applications repository](#)

Let's repeat it here:

```

1 from random import normalvariate
2 import matplotlib.pyplot as plt
3 ts_length = 100
4 epsilon_values = [] # An empty list
5 for i in range(ts_length):

```

```

6     e = normalvariate(0, 1)
7     epsilon_values.append(e)
8 plt.plot(epsilon_values, 'b-')
9 plt.show()

```

In brief,

- Lines 1–2 use the Python `import` keyword to pull in functionality from external libraries
- Line 3 sets the desired length of the time series
- Line 4 creates an empty list called `epsilon_values` that will store the  $\epsilon_t$  values as we generate them
- Line 5 tells the Python interpreter that it should cycle through the block of indented lines (lines 6–7) `ts_length` times before continuing to line 8
  - Lines 6–7 draw a new value  $\epsilon_t$  and append it to the end of the list `epsilon_values`
- Lines 8–9 generate the plot and display it to the user

Let's now break this down and see how the different parts work

**Import Statements** First we'll look at how to import functionality from outside your program, as in lines 1–2

**Modules** Consider the line `from random import normalvariate`

Here `random` is a *module*, which is just a file containing Python code

The statement `from random import normalvariate` causes the Python interpreter to

- run the code in a file called `random.py` that was placed in your filesystem when you installed Python
- make the function `normalvariate` defined in that file available for use in your program

If you want to import more attributes you can use a comma separated list, like so:

```

In [4]: from random import normalvariate, uniform

In [5]: normalvariate(0, 1)
Out[5]: -0.38430990243287594

In [6]: uniform(-1, 1)
Out[6]: 0.5492316853602877

```

Alternatively, you can use the following syntax:

```

In [1]: import random

In [2]: random.normalvariate(0, 1)
Out[2]: -0.12451500570438317

```

```
In [3]: random.uniform(-1, 1)
Out[3]: 0.35121616197003336
```

After importing the module itself, we can access anything defined within via `module_name.attribute_name` syntax

**Packages** Now consider the line `import matplotlib.pyplot as plt`

Here `matplotlib` is a Python *package*, and `pyplot` is a *subpackage* of `matplotlib`

Packages are used by developers to organize a number of related Python files (modules)

A package is just a directory containing

1. modules
2. a file called `__init__.py` that specifies what code will be run when we type `import package_name`

Subpackages are the same, except that they are subdirectories of a package directory

So `import matplotlib.pyplot as plt` runs the `__init__.py` file in the directory `matplotlib/pyplot` and makes the attributes specified in that file available to us

The keyword `as` in `import matplotlib.pyplot as plt` just lets us access these attributes via a simpler name

**Lists** Next let's 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

```
In [7]: x = [10, 'foo', False] # We can include heterogeneous data inside a list
In [8]: type(x)
Out[8]: list
```

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

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

```
In [9]: x
Out[9]: [10, 'foo', False]

In [10]: x.append(2.5)

In [11]: x
Out[11]: [10, 'foo', False, 2.5]
```

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

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

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

- String objects have `string methods`, list objects have `list methods`, etc.

Another useful list method is `pop()`

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

In [13]: x.pop()
Out[13]: 2.5

In [14]: x
Out[14]: [10, 'foo', False]
```

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

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

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

In [16]: x[0]
Out[16]: 10

In [17]: x[1]
Out[17]: 'foo'
```

Returning to `test_program_1.py` above, we actually create a second list besides `epsilon_values`

In particular, line 5 calls the `range()` function, which creates sequential lists of integers

```
In [18]: range(4)
Out[18]: [0, 1, 2, 3]

In [19]: range(5)
Out[19]: [0, 1, 2, 3, 4]
```

**The For Loop** Now let's consider the `for` loop in `test_program_1.py`, which we repeat here for convenience, along with the line that follows it

```
for i in range(ts_length):
    e = normalvariate(0, 1)
    epsilon_values.append(e)
plt.plot(epsilon_values, 'b-')
```

The `for` loop causes Python to execute the two indented lines a total of `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 particular, the fact that indentation decreases after line `epsilon_values.append(e)` tells 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 occurring inside loops, `if` clauses, function definitions, etc.) are delimited by indentation

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

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

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

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

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

**Tabs vs Spaces** One small “gotcha” here is the mixing of tabs and spaces, which often leads to errors

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

You can use your Tab key to insert 4 spaces, but you need to make sure 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 `test_program_1.py` to use a `while` loop instead

In Python, the `while` loop syntax is as shown in the file `test_program_2.py` below

```

1 from random import normalvariate
2 import matplotlib.pyplot as plt
3 ts_length = 100
4 epsilon_values = []
5 i = 0
6 while i < ts_length:
7     e = normalvariate(0, 1)
8     epsilon_values.append(e)
9     i = i + 1
10 plt.plot(epsilon_values, 'b-')
11 plt.show()
```

The output of `test_program_2.py` is identical to `test_program_1.py` above (modulo randomness)

Comments:

- The code block for the `while` loop is lines 7–9, 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 `test_program_3.py`

```

1 from random import normalvariate
2 import matplotlib.pyplot as plt
3
```

```

4
5 def generate_data(n):
6     epsilon_values = []
7     for i in range(n):
8         e = normalvariate(0, 1)
9         epsilon_values.append(e)
10    return epsilon_values
11
12 data = generate_data(100)
13 plt.plot(data, 'b-')
14 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()`, where the definition spans lines 4–9

- `def` on line 4 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`
- Lines 5–9 are a code block called the *function body*—in this case it creates an iid list of random draws using the same logic as before
- Line 9 indicates that the list `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)` in line 12, it executes the function body (lines 5–9) with `n` set equal to 100.

The net result is that the name `data` on the left-hand side of line 12 is set equal 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 in `test_program_4.py` by adding the argument `generator_type` to `generate_data()`

```

1 from random import normalvariate, uniform
2 import matplotlib.pyplot as plt
3
4
5 def generate_data(n, generator_type):
6     epsilon_values = []
7     for i in range(n):
8         if generator_type == 'U':
9             e = uniform(0, 1)
10        else:
11            e = normalvariate(0, 1)
12        epsilon_values.append(e)

```

```

13     return epsilon_values
14
15 data = generate_data(100, 'U')
16 plt.plot(data, 'b-')
17 plt.show()

```

Comments:

- Hopefully the syntax of the if/else clause is self-explanatory, with indentation again delimiting the extent of the code blocks
- 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 two ways that we can simplify test\_program\_4

First, Python accepts the following conditional assignment syntax

```

In [20]: x = -10

In [21]: s = 'negative' if x < 0 else 'nonnegative'

In [22]: s
Out[22]: 'negative'

```

which leads us to test\_program\_5.py

```

1 from random import normalvariate, uniform
2 import matplotlib.pyplot as plt
3
4
5 def generate_data(n, generator_type):
6     epsilon_values = []
7     for i in range(n):
8         e = uniform(0, 1) if generator_type == 'U' else normalvariate(0, 1)
9         epsilon_values.append(e)
10    return epsilon_values
11
12 data = generate_data(100, 'U')
13 plt.plot(data, 'b-')
14 plt.show()

```

Second, and more importantly, we can get rid of the conditionals all together by just passing the desired generator type *as a function*

To understand this, consider test\_program\_6.py

```

1 from random import uniform
2 import matplotlib.pyplot as plt
3
4

```

```

5 def generate_data(n, generator_type):
6     epsilon_values = []
7     for i in range(n):
8         e = generator_type(0, 1)
9         epsilon_values.append(e)
10    return epsilon_values
11
12 data = generate_data(100, uniform)
13 plt.plot(data, 'b-')
14 plt.show()

```

The only lines that have changed here are lines 7 and 11

In line 11, when we call the function `generate_data()`, we pass `uniform` as the second argument

The object `uniform` is in fact a **function**, defined in the `random` module

```
In [23]: from random import uniform
```

```
In [24]: uniform(0, 1)
```

```
Out[24]: 0.2981045489306786
```

When the function call `generate_data(100, uniform)` on line 11 is executed, Python runs the code block on lines 5–9 with `n` equal to 100 and the name `generator_type` “bound” to the function `uniform`

- While these lines are executed, the names `generator_type` and `uniform` are “synonyms”, and can be used in identical ways

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

```
In [25]: max(7, 2, 4)    # max() is a built-in Python function
Out[25]: 7
```

```
In [26]: m = max
```

```
In [27]: m(7, 2, 4)
```

```
Out[27]: 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 do in line 11

**List Comprehensions** Now is probably a good time to tell you that we can simplify the code for generating the list of random draws considerably by using something called a *list comprehension*

List comprehensions are an elegant Python tool for creating lists

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

```
In [28]: animals = ['dog', 'cat', 'bird']

In [29]: plurals = [animal + 's' for animal in animals]

In [30]: plurals
Out[30]: ['dogs', 'cats', 'birds']
```

Here's another example

```
In [31]: range(8)
Out[31]: [0, 1, 2, 3, 4, 5, 6, 7]

In [32]: doubles = [2 * x for x in range(8)]

In [33]: doubles
Out[33]: [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(0, 1)
    epsilon_values.append(e)
```

into

```
epsilon_values = [generator_type(0, 1) for i in range(n)]
```

**Using the Scientific Libraries** As discussed at the start of the lecture, our example is somewhat contrived

In practice we would use the scientific libraries, which can generate large arrays of independent random draws much more efficiently

For example, try

```
In [34]: from numpy.random import randn

In [35]: epsilon_values = randn(5)

In [36]: epsilon_values
Out[36]: array([-0.15591709, -1.42157676, -0.67383208, -0.45932047, -0.17041278])
```

We'll discuss these scientific libraries a bit later on

## 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 random import uniform`, write a function `binomial_rv` such that `binomial_rv(n, p)` generates one draw of  $Y$

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

**Exercise 3** Compute an approximation to  $\pi$  using Monte Carlo. Use no imports besides

```
from random import uniform
from math import sqrt
```

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  $U_1, \dots, U_n$  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 random import uniform`

**Exercise 5** Your next task is to simulate and plot the correlated time series

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

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

In your solution, restrict your import statements to

```
import matplotlib.pyplot as plt
from random import normalvariate
```

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

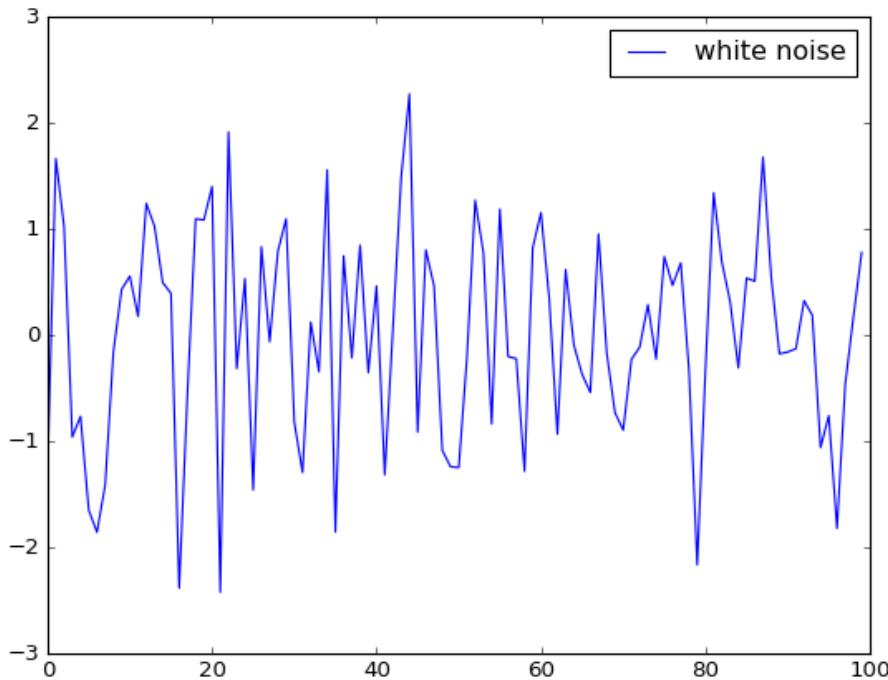
**Exercise 6** To do the next exercise, you will need to know how to produce a plot legend

The following example should be sufficient to convey the idea

```
from pylab import plot, show, legend
from random import normalvariate

x = [normalvariate(0, 1) for i in range(100)]
plot(x, 'b-', label="white noise")
legend()
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

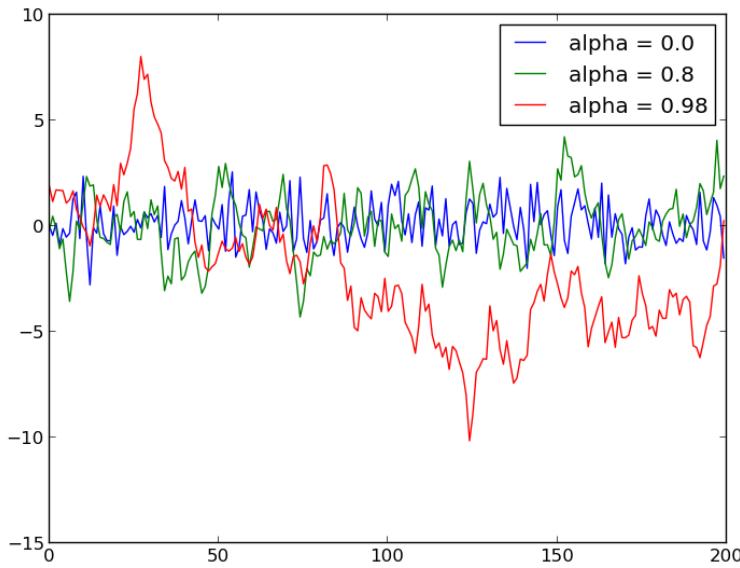
(The figure nicely illustrates how time series with the same one-step-ahead conditional volatilities, as these three processes have, can have very different unconditional volatilities.)

In your solution, please restrict your import statements to

```
import matplotlib.pyplot as plt
from random import normalvariate
```

Also, use a for loop to step through the  $\alpha$  values

Important hints:



- If you call the `plot()` function multiple times before calling `show()`, all of the lines you produce will end up on the same figure
  - And if you omit the argument '`b-`' to the `plot` function, Matplotlib will automatically select different colors for each line
- The expression '`foo`' + `str(42)` evaluates to '`foo42`'

## Solutions

[Solution notebook](#)

## 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

```
In [1]: x = True  
  
In [2]: y = 100 < 10    # Python evaluates expression on right and assigns it to y  
  
In [3]: y  
Out[3]: False
```

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

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

```
In [5]: x + y
Out[5]: 1

In [6]: x * y
Out[6]: 0

In [7]: True + True
Out[7]: 2

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

In [9]: sum(bools)
Out[9]: 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

```
In [1]: a, b = 1, 2

In [2]: c, d = 2.5, 10.0

In [3]: type(a)
Out[3]: int

In [4]: type(c)
Out[4]: float
```

Computers distinguish between the two because, while floats are more informative, integral 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:

```
In [5]: 1 / 2      # Integer division in Python 2.x
Out[5]: 0

In [6]: 1.0 / 2.0  # Floating point division
Out[6]: 0.5

In [7]: 1.0 / 2    # Floating point division
Out[7]: 0.5
```

If you're using Python 3.x this is no longer a concern

Complex numbers are another primitive data type in Python

```
In [10]: x = complex(1, 2)
In [11]: y = complex(2, 1)
In [12]: x * y
Out[12]: 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

```
In [13]: x = ('a', 'b') # Round brackets instead of the square brackets
In [14]: x = 'a', 'b' # Or no brackets at all---the meaning is identical
In [15]: x
Out[15]: ('a', 'b')
In [16]: type(x)
Out[16]: tuple
```

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

Lists are mutable while tuples are not

```
In [17]: x = [1, 2] # Lists are mutable
In [18]: x[0] = 10 # Now x = [10, 2], so the list has "mutated"
In [19]: x = (1, 2) # Tuples are immutable
In [20]: x[0] = 10 # Trying to mutate them produces an error
-----
TypeError                                     Traceback (most recent call last)
<ipython-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

```
In [21]: integers = (10, 20, 30)
In [22]: x, y, z = integers
```

```
In [23]: x
Out[23]: 10
```

```
In [24]: y
Out[24]: 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,

```
In [14]: a = [2, 4, 6, 8]
```

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

```
In [16]: a[1:3]
Out[16]: [4, 6]
```

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

Negative numbers are also permissible

```
In [17]: a[-2:] # Last two elements of the list
Out[17]: [6, 8]
```

The same slice notation works on tuples and strings

```
In [19]: s = 'foobar'
```

```
In [20]: s[-3:] # Select the last three elements
Out[20]: 'bar'
```

**Sets and Dictionaries** Two other container types we should mention before moving on are `sets` and `dictionaries`

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

```
In [25]: d = {'name': 'Frodo', 'age': 33}
```

```
In [26]: type(d)
Out[26]: dict
```

```
In [27]: d['age']
Out[27]: 33
```

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

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

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

```
In [28]: s1 = {'a', 'b'}
In [29]: type(s1)
Out[29]: set
In [30]: s2 = {'b', 'c'}
In [31]: s1.issubset(s2)
Out[31]: False
In [32]: s1.intersection(s2)
Out[32]: set(['b'])
```

The `set()` function creates sets from sequences

```
In [33]: s3 = set(['foo', 'bar', 'foo'])
In [34]: s3
Out[34]: 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`

```
In [1]: import math
In [2]: math.sqrt(4)
Out[2]: 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

```
In [3]: from math import *
In [4]: sqrt(4)
Out[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

```
In [35]: f = open('newfile.txt', 'w')      # Open 'newfile.txt' for writing
In [36]: f.write('Testing\n')                # Here '\n' means new line
In [37]: f.write('Testing again')
In [38]: 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())
```

(In IPython or a Jupyter notebook, `%pwd` or `pwd` should also work)

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

(In IPython, 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

```
In [39]: f = open('newfile.txt', 'r')

In [40]: out = f.read()

In [41]: out
Out[41]: 'Testing\nTesting again'

In [42]: print(out)
Out[42]:
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

```
In [43]: 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 IPython, 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 be 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:

```

1 data_file = open('us_cities.txt', 'r')
2 for line in data_file:
3     city, population = line.split(':')           # Tuple unpacking
4     city = city.title()                         # Capitalize city names
5     population = '{0:,}'.format(int(population)) # Add commas to numbers
6     print(city.ljust(15) + population)
7 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 {} is {}'.format(country, city))
```

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

```
In [1]: names = ['Tom', 'John']
In [2]: marks = ['E', 'F']
In [3]: dict(zip(names, marks))
Out[3]: {'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[{}]={}".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

```
In [44]: x, y = 1, 2
In [45]: x < y
Out[45]: True
In [46]: x > y
Out[46]: False
```

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

```
In [47]: 1 < 2 < 3
Out[47]: True
In [48]: 1 <= 2 <= 3
Out[48]: True
```

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

```
In [49]: x = 1      # Assignment
In [50]: x == 2    # Comparison
Out[50]: False
```

For “not equal” use !=

```
In [51]: 1 != 2
Out[51]: True
```

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

```
In [52]: x = 'yes' if 42 else 'no'
In [53]: x
Out[53]: 'yes'

In [54]: x = 'yes' if [] else 'no'
In [55]: x
Out[55]: 'no'
```

What’s going on here? The rule is:

- Expressions that evaluate to zero, empty sequences/containers (strings, lists, etc.) and None are equivalent to False
- All other values are equivalent to True

**Combining Expressions** We can combine expressions using and, or and not

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

```
In [56]: 1 < 2 and 'f' in 'foo'
Out[56]: True

In [57]: 1 < 2 and 'g' in 'foo'
Out[57]: False

In [58]: 1 < 2 or 'g' in 'foo'
Out[58]: True

In [59]: not True
Out[59]: False

In [60]: not not True
Out[60]: 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

```
In [61]: max(19, 20)
Out[61]: 20
```

```
In [62]: range(4)
Out[62]: [0, 1, 2, 3]
```

```
In [63]: str(22)
Out[63]: '22'
```

```
In [64]: type(22)
Out[64]: int
```

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

```
In [65]: bools = False, True, True
In [66]: all(bools)  # True if all are True and False otherwise
Out[66]: False
In [67]: any(bools)  # False if all are False and True otherwise
Out[67]: 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
- Any object can be passed to a function as an argument, including other functions
- Functions can be (and often are) defined inside 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 it has been run in the IPython shell, the docstring is available as follows

```
In [1]: run temp.py

In [2]: f?
Type:     function
String Form:<function f at 0x2223320>
File:      /home/john/temp/temp.py
Definition: f(x)
Docstring: This function squares its argument

In [3]: 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

```
In [68]: from scipy.integrate import quad
In [69]: quad(lambda x: x**3, 0, 2)
Out[69]: (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

```
In [71]: f(2, coefficients=(0, 0))
Out[71]: 0

In [72]: f(2)    # Use default values (1, 1)
Out[72]: 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

**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

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n = \sum_{i=0}^n a_i x^i \quad (1.1)$$

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

[Solution notebook](#)

## 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.

```
In [1]: x = [1, 5, 4]
In [2]: x.sort()
In [3]: x
Out[3]: [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

```
In [4]: x = [1, 5, 4]
```

```
In [5]: x.sort()
```

```
In [6]: x.__class__
```

```
Out[6]: 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

```
In [1]: class Consumer:
    ...:     pass
    ...:

In [2]: c1 = Consumer()  # Create an instance

In [3]: c1.wealth = 10

In [4]: c1.wealth
Out[4]: 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` in the `applications` repository

```
class Consumer:

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

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

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

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

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, assuming `consumer.py` is in your present working directory

```
In [1]: run consumer.py

In [2]: c1 = Consumer(10)  # Create instance with initial wealth 10

In [3]: c1.spend(5)

In [4]: c1.wealth
Out[4]: 5

In [5]: c1.earn(15)

In [6]: c1.spend(100)
Insufficient funds
```

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

```
In [2]: c1 = Consumer(10)

In [3]: c2 = Consumer(12)

In [4]: c2.spend(4)

In [5]: c2.wealth
Out[5]: 8

In [6]: c1.wealth
Out[6]: 10
```

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

```
In [7]: c1.__dict__
Out[7]: {'wealth': 10}

In [8]: c2.__dict__
Out[8]: {'wealth': 8}
```

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

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

The rules with `self` are that

- Any instance data should be prepended with `self`
  - e.g., the `earn` method references `self.wealth` rather than just `wealth`

- Any method defined within the class should have *self* as its first argument
  - e.g., *def earn(self, y)* rather than just *def earn(y)*
- Any method referenced within the class should be called as *self.method\_name*

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

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

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

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

```
In [1]: run consumer.py # Read class def, build class object Consumer

In [2]: 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

```
In [2]: c1 = Consumer(10)

In [3]: c1.earn(10)

In [4]: c1.wealth
Out[4]: 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

$$k_{t+1} = \frac{szk_t^\alpha + (1 - d)k_t}{1 + n} \quad (1.2)$$

Here

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

The **steady state** of the model is the  $k$  that solves (1.2) when  $k_{t+1} = k_t = k$

While `QuantEcon.applications` already has some relatively sophisticated code for dealing with this model, here we'll create something more basic for illustrative purposes

You can find the file below in `solow.py` in the `applications` repository

```
"""
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:
    """
    Implements the Solow growth model with update rule

    .. math::
        k_{t+1} = \frac{s z k_t^{\alpha} (1 + n)}{1 + d} + 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 (1.2)
- The *update* method uses *h* to update capital as per (1.2)
  - Notice how inside *update* the reference to the local method *h* is *self.h*

The methods *steady\_state* and *generate\_sequence* are fairly self explanatory

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

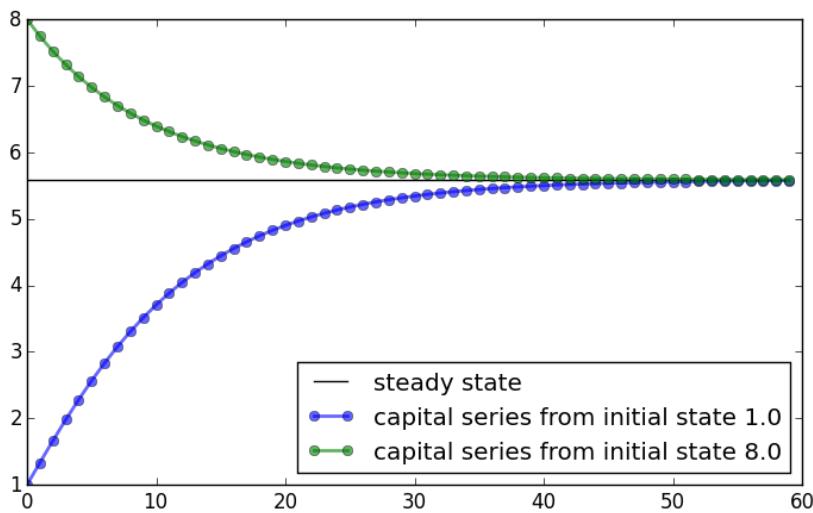
```

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 equilibrium 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

```

In [1]: run market.py

In [2]: baseline_params = 15, .5, -2, .5, 3

In [3]: m = Market(*baseline_params)

```

```
In [4]: print("equilibrium price = ", m.price())
equilibrium price = 18.5

In [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
from market import Market

# 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

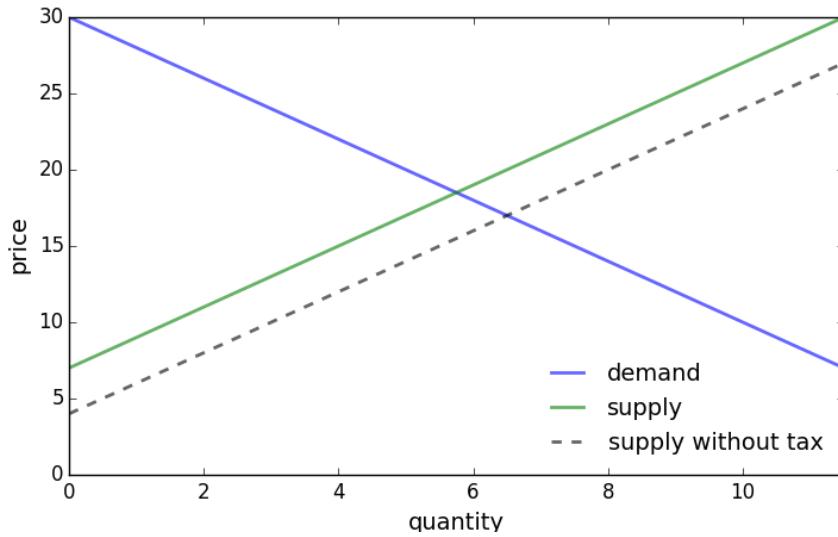
The next program provides a function that

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

```
from market import Market

def deadw(m):
    "Computes deadweight loss for market m."
    # == Create analogous market with no tax == #
    m_no_tax = Market(m.ad, m.bd, m.az, m.bz, 0)
    # == Compare surplus, return difference == #
    surp1 = m_no_tax.consumer_surp() + m_no_tax.producer_surp()
    surp2 = m.consumer_surp() + m.producer_surp() + m.taxrev()
    return surp1 - surp2
```

Here's an example of usage



```
In [5]: run market_deadweight.py
In [6]: baseline_params = 15, .5, -2, .5, 3
In [7]: m = Market(*baseline_params)
In [8]: deadw(m) # Show deadweight loss
Out[8]: 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

$$x_{t+1} = rx_t(1 - x_t), \quad x_0 \in [0, 1], \quad r \in [0, 4] \quad (1.3)$$

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

```

In [1]: run chaos_class.py

In [2]: ch = Chaos(0.1, 4.0) # x0 = 0.1 and r = 0.4

In [3]: ch.generate_sequence(5) # First 5 iterates
Out[3]: [0.1, 0.3600000000000004, 0.9216, 0.2890137600000006, 0.8219392261226498]

```

This piece of code plots a longer trajectory

```

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

from chaos_class import Chaos
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

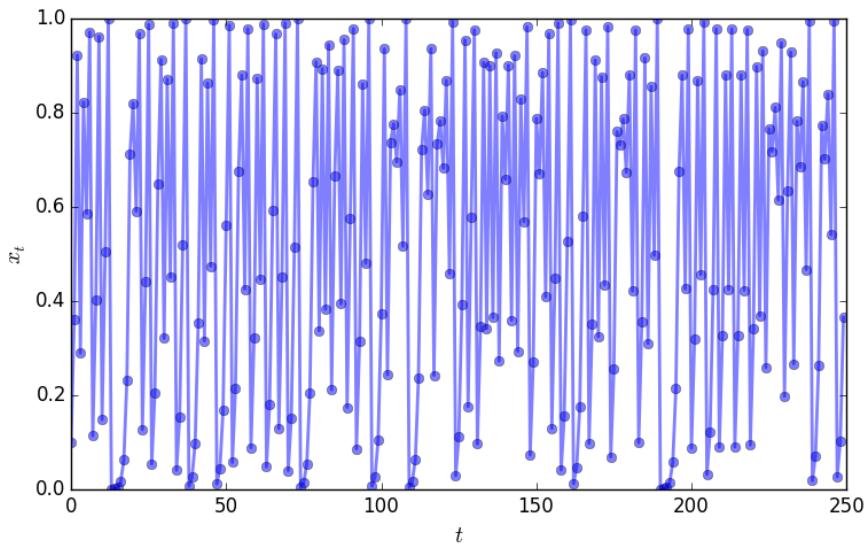
```

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

from chaos_class import Chaos
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ch = Chaos(0.1, 4)
r = 2.5

```



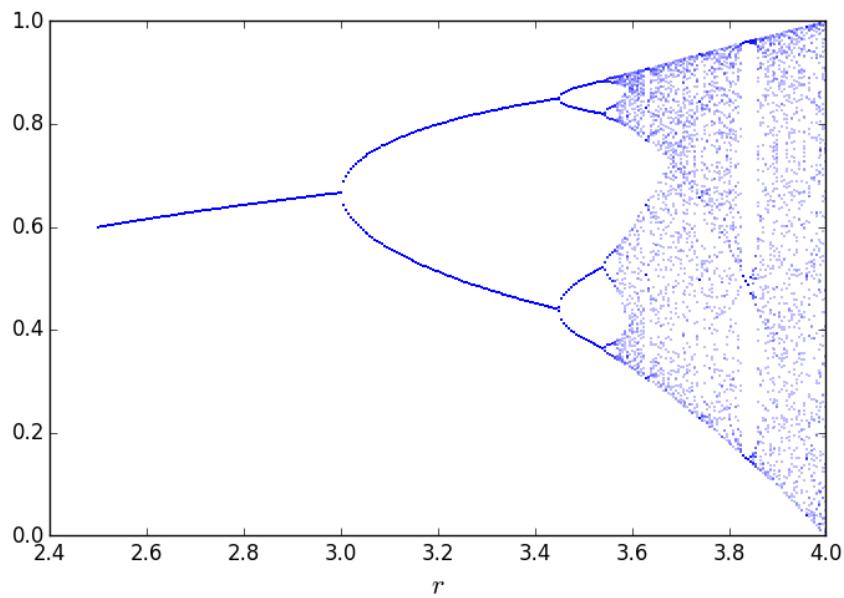
```

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 (1.3)

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

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

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

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

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

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

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

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

## Special Methods

Python provides special methods with which some neat tricks can be performed

For example, recall that lists and tuples have a notion of length, and that this length can be queried via the `len` function

```
In [21]: x = (10, 20)
```

```
In [22]: len(x)
Out[22]: 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

```
In [23]: f = Foo()

In [24]: len(f)
Out[24]: 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

```
In [25]: f = Foo()

In [26]: f(8)  # Exactly equivalent to f.__call__(8)
Out[26]: 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

$$F_n(x) := \frac{1}{n} \sum_{i=1}^n \mathbf{1}\{X_i \leq x\} \quad (x \in \mathbb{R}) \quad (1.4)$$

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

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

Implement  $F_n$  as a class called 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)

```
In [28]: from random import uniform

In [29]: samples = [uniform(0, 1) for i in range(10)]

In [30]: F = ECDF(samples)

In [31]: F(0.5)  # Evaluate ecdf at x = 0.5
Out[31]: 0.29

In [32]: F.observations = [uniform(0, 1) for i in range(1000)]

In [33]: F(0.5)
Out[33]: 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

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_Nx^N = \sum_{n=0}^N a_nx^n \quad (x \in \mathbb{R}) \quad (1.5)$$

The instance data for the class `Polynomial` will be the coefficients (in the case of (1.5), the numbers  $a_0, \dots, a_N$ )

Provide methods that

1. Evaluate the polynomial (1.5), returning  $p(x)$  for any  $x$
2. Differentiate the polynomial, replacing the original coefficients with those of its derivative  $p'$

Avoid using any `import` statements

## Solutions

[Solution notebook](#)

# 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

```
In [1]: 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

```
In [2]: s = 'This is a string'

In [3]: type(s)
Out[3]: str

In [4]: x = 42    # Now let's create an integer

In [5]: type(x)
Out[5]: int
```

The type of an object matters for many expressions

For example, the addition operator between two strings means concatenation

```
In [6]: '300' + 'cc'
Out[6]: '300cc'
```

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

```
In [7]: 300 + 400
Out[7]: 700
```

Consider the following expression

```
In [8]: '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-1-9b7dfffd27f2d> 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,

```
In [9]: int('300') + 400    # To add as numbers, change the string to an integer
Out[9]: 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

```
In [10]: x = 42

In [11]: x
Out[11]: 42

In [12]: x.imag
Out[12]: 0

In [13]: x.__class__
Out[13]: 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

```
In [14]: y = 2.5
In [15]: z = 2.5
In [16]: id(y)
Out[16]: 166719660
In [17]: id(z)
Out[17]: 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)

```
In [18]: x = ['foo', 'bar']
In [19]: callable(x.append)
Out[19]: True
In [20]: callable(x.__doc__)
Out[20]: False
```

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

```
In [21]: x = ['a', 'b']
In [22]: x.append('c')
In [23]: s = 'This is a string'
In [24]: s.upper()
Out[24]: 'THIS IS A STRING'
In [25]: s.lower()
Out[25]: 'this is a string'
```

```
In [26]: s.replace('This', 'That')
Out[26]: 'That is a string'
```

A great deal of Python functionality is organized around method calls

For example, consider the following piece of code

```
In [27]: x = ['a', 'b']

In [28]: x[0] = 'aa' # Item assignment using square bracket notation

In [29]: x
Out[29]: ['aa', 'b']
```

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

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

```
In [30]: x = ['a', 'b']

In [31]: x.__setitem__(0, 'aa') # Equivalent to x[0] = 'aa'

In [32]: x
Out[32]: ['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

```
In [33]: def f(x): return x**2

In [34]: f
Out[34]: <function __main__.f>

In [35]: type(f)
Out[35]: function

In [36]: id(f)
Out[36]: 3074342220L

In [37]: f.__name__
Out[37]: '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

```
In [38]: import math
In [39]: id(math)
Out[39]: 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*

```
In [1]: f = open('us_cities.txt')
In [2]: f.__next__()
Out[2]: 'new york: 8244910\n'
In [3]: f.__next__()
Out[3]: 'los angeles: 3819702\n'
```

We see that file objects do indeed have a `__next__` method, and that calling this method returns the next line in the file

The `next` method can also be accessed via the builtin function `next()`, which directly calls this method

```
In [4]: next(f)
Out[4]: 'chicago: 2707120 \n'
```

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

```
In [43]: e = enumerate(['foo', 'bar'])
In [44]: next(e)
Out[44]: (0, 'foo')
In [45]: next(e)
Out[45]: (1, 'bar')
```

as are the reader objects from the `csv` module

```
In [46]: from csv import reader

In [47]: f = open('test_table.csv', 'r')

In [48]: nikkei_data = reader(f)

In [49]: next(nikkei_data)
Out[49]: ['Date', 'Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close']

In [50]: next(nikkei_data)
Out[50]: ['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

```
In [7]: for i in ['spam', 'eggs']:
    ...:     print(i)
    ...:
spam
eggs
```

So does that mean that a list is an iterator?

The answer is no:

```
In [16]: type(x)
Out[16]: list

In [17]: 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

```
In [59]: x = ['foo', 'bar']

In [60]: type(x)
Out[60]: list

In [61]: y = iter(x)

In [62]: type(y)
Out[62]: list_iterator

In [63]: next(y)
Out[63]: 'foo'

In [64]: next(y)
Out[64]: 'bar'

In [65]: 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

```
In [66]: 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

```
In [67]: x = [10, -10]

In [68]: max(x)
Out[68]: 10

In [69]: y = iter(x)

In [70]: type(y)
Out[70]: listiterator

In [71]: max(y)
Out[71]: 10
```

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

```
In [72]: x = [10, -10]

In [73]: y = iter(x)

In [74]: max(y)
Out[74]: 10

In [75]: 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

```
In [76]: x = 42
```

We now know that when this statement is executed, Python creates an object of type `int` in your computer's memory, containing

- the value 42

- some associated attributes

But what is `x` itself?

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

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

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

```
In [77]: def f(string):      # Create a function called f
....:     print(string)    # that prints any string it's passed

In [78]: g = f

In [79]: id(g) == id(f)
Out[79]: True

In [80]: g('test')
Out[80]: test
```

In the first step, a function object is created, and the name `f` is bound to it

After binding the name `g` to the same object, we can use it anywhere we would use `f`

What happens when the number of names bound to an object goes to zero?

Here's an example of this situation, where the name `x` is first bound to one object and then rebound to another

```
In [81]: x = 'foo'

In [82]: id(x)
Out[82]: 164994764

In [83]: 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

```
In [84]: 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

```
In [85]: import math2
```

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

```
In [86]: import math
```

Both of these modules have an attribute called `pi`

```
In [87]: math.pi
Out[87]: 3.1415926535897931
```

```
In [88]: math2.pi
Out[88]: 'foobar'
```

These two different bindings of `pi` exist in different namespaces, each one implemented as a dictionary

We can look at the dictionary directly, using `module_name.__dict__`

```
In [89]: import math
In [90]: math.__dict__
Out[90]: {'pow': <built-in function pow>, ..., 'pi': 3.1415926535897931,...} # Edited output
In [91]: import math2
In [92]: math2.__dict__
Out[92]: {..., '__file__': 'math2.py', 'pi': 'foobar',...} # Edited output
```

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

```
In [93]: math.pi
Out[93]: 3.1415926535897931
```

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

```
In [94]: math.__dict__['pi'] == math.pi
Out[94]: 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)`

```
In [95]: vars(math)
Out[95]: {'pow': <built-in function pow>, ...}
```

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

```
In [96]: dir(math)
Out[96]: ['__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

```
In [97]: print(math.__doc__)
This module is always available. It provides access to the
mathematical functions defined by the C standard.
```

```
In [98]: 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

```
In [99]: 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

```
In [1]: import mod # Standard import
mod

In [2]: run mod.py # Run interactively
__main__
```

In the second case, the code is executed as part of `__main__`, so `__name__` is equal to `__main__`

To see the contents of the namespace of `__main__` we use `vars()` rather than `vars(__main__)`

If you do this in IPython, you will see a whole lot of variables that IPython needs, and has initialized when you started up your session

If you prefer to see only the variables you have initialized, use `whos`

```
In [3]: x = 2
In [4]: y = 3
In [5]: import numpy as np
In [6]: whos
          Variable      Type      Data/Info
          -----
          np            module   <module 'numpy' from '/usr/local/lib/python2.7/dist-packages/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`

```
In [7]: 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

```
In [1]: def f(x):
....:     a = 2
....:     print(locals())
....:     return a * x
....:
```

Now let's call the function

```
In [2]: 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 their own namespace called `__builtins__`

```
In [12]: dir()
Out[12]: [..., '__builtins__', '__doc__', ...] # Edited output

In [13]: dir(__builtins__)
Out[13]: [... 'iter', 'len', 'license', 'list', 'locals', ...] # Edited output
```

We can access elements of the namespace as follows

```
In [14]: __builtins__.max
Out[14]: <built-in function max>
```

But `__builtins__` is special, because we can always access them directly as well

```
In [15]: max
Out[15]: <built-in function max>

In [16]: __builtins__.max == max
Out[16]: 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?

```
In [17]: run test.py
a = 0 y = 11
```

```
In [18]: 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  $y_1, \dots, y_n$  is defined as

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

This can be calculated in NumPy using `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

```
In [19]: def var(y):
....:     n = len(y)
....:     assert n > 1, 'Sample size must be greater than one.'
....:     return np.sum((y - y.mean())**2) / float(n-1)
....:
```

If we run this with an array of length one, the program will terminate and print our error message

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

<ipython-input-19-cefaefac3555> 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

```
In [43]: 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

```
In [44]: 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

```
In [45]: x1 = y1  
  
-----  
NameError                                     Traceback (most recent call last)  
<ipython-input-23-142e0509fdb6> in <module>()  
----> 1 x1 = y1  
  
NameError: name 'y1' is not defined
```

And another

```
In [46]: '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

```
In [47]: X = []
In [48]: 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 occasion, the interpreter informs us of the error type

- `NameError`, `TypeError`, `IndexError`, `ZeroDivisionError`, etc.

In Python, these errors are called *exceptions*

**Catching Exceptions** We can catch and deal with exceptions using `try – except` blocks

Here's a simple example

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

When we call `f` we get the following output

```
In [50]: f(2)
Out[50]: 0.5

In [51]: f(0)
          Error: division by zero. Returned None

In [52]: 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

```
In [54]: f(2)
Out[54]: 0.5

In [55]: f(0)
          Error: Division by zero. Returned None

In [56]: 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

```
In [58]: f(2)
Out[58]: 0.5

In [59]: f(0)
          Error: Unsupported operation. Returned None

In [60]: 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*

Suppose further that this is not what we want because it causes other problems that are hard to pick up

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(object):

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

    # Some other functionality, details omitted
```

One potential problem we might have here is that a user alters one of these variables but not the other

```
In [2]: car = Car()

In [3]: car.miles
Out[3]: 1000

In [4]: car.kms
Out[4]: 1610.0

In [5]: car.miles = 6000
```

```
In [6]: car.kms
Out[6]: 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(object):

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

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

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

    def get_miles(self):
        return self._miles

    def get_kms(self):
        return self._kms

    miles = property(get_miles, set_miles)
    kms = property(get_kms, set_kms)
```

First let's check that we get the desired behavior

```
In [8]: car = Car()

In [9]: car.miles
Out[9]: 1000

In [10]: car.miles = 6000

In [11]: car.kms
Out[11]: 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(object):

    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:

```
In [1]: singular = ('dog', 'cat', 'bird')

In [2]: type(singular)
Out[2]: tuple

In [3]: plural = [string + 's' for string in singular]

In [4]: plural
Out[4]: ['dogs', 'cats', 'birds']

In [5]: type(plural)
Out[5]: list
```

And here is the generator expression

```
In [6]: singular = ('dog', 'cat', 'bird')

In [7]: plural = (string + 's' for string in singular)

In [8]: type(plural)
Out[8]: generator

In [9]: next(plural)
Out[9]: 'dogs'

In [10]: next(plural)
Out[10]: 'cats'

In [11]: next(plural)
Out[11]: 'birds'
```

Since `sum()` can be called on iterators, we can do this

```
In [12]: sum((x * x for x in range(10)))
Out[12]: 285
```

The function `sum()` calls `next()` to get the items, adds successive terms

In fact, we can omit the outer brackets in this case

```
In [13]: sum(x * x for x in range(10))
Out[13]: 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

```
In [15]: type(f)
Out[15]: function

In [16]: gen = f()

In [17]: gen
Out[17]: <generator object f at 0x3b66a50>

In [18]: next(gen)
Out[18]: 'start'

In [19]: next(gen)
Out[19]: 'middle'

In [20]: next(gen)
Out[20]: 'end'

In [21]: 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

```
In [24]: g
Out[24]: <function __main__.g>

In [25]: gen = g(2)

In [26]: type(gen)
Out[26]: generator

In [27]: next(gen)
Out[27]: 2

In [28]: next(gen)
Out[28]: 4

In [29]: next(gen)
Out[29]: 16

In [30]: next(gen)

-----
StopIteration                                                 Traceback (most recent call last)
<ipython-input-32-b2c61ce5e131> in <module>()
      1 gen.next()
-----> 1 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  $\text{binomial}(n, 0.5)$

One way to do it is as follows

```
In [32]: n = 10000000
In [33]: draws = [random.uniform(0, 1) < 0.5 for i in range(n)]
In [34]: 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

```
In [35]: n = 1000000000
In [36]: 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

```
In [39]: n = 10000000
In [40]: draws = f(n)
```

```
In [41]: draws
Out[41]: <generator object at 0xb7d8b2cc>
```

```
In [42]: sum(draws)
Out[42]: 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

$$x_{t+1} = 2x_t, \quad x_0 = 1 \quad (1.6)$$

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 its own *frame* in the *stack*

- a frame is where the local variables of a given function call are held
- **stack is memory used to process function calls**
  - a First In Last Out (FILO) queue

This example is somewhat contrived, since the first (iterative) solution would usually be preferred to the recursive solution

We'll meet less contrived applications of recursion later on

## Exercises

**Exercise 1** The Fibonacci numbers are defined by

$$x_{t+1} = x_t + x_{t-1}, \quad x_0 = 0, \quad x_1 = 1 \quad (1.7)$$

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

[Solution notebook](#)

## 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 the latest Python 3.5 version 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:

```
In [1]: import numpy as np  
  
In [2]: x = np.random.uniform(0, 1, size=1000000)  
  
In [3]: x.mean()  
Out[3]: 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`

```
In [1]: import numpy as np  
  
In [2]: a = np.zeros(3)  
  
In [3]: a  
Out[3]: array([ 0.,  0.,  0.])
```

```
In [4]: type(a)
Out[4]: numpy.ndarray
```

NumPy arrays are somewhat like native Python lists, except that

- Data *must be homogeneous* (all elements of the same type)
- These types must be one of the data types (**dtypes**) provided by NumPy

The most important of these dtypes are:

- float64: 64 bit floating point number
- int64: 64 bit integer
- bool: 8 bit True or False

There are also dtypes to represent complex numbers, unsigned integers, etc

On modern machines, the default dtype for arrays is float64

```
In [7]: a = np.zeros(3)

In [8]: type(a[0])
Out[8]: numpy.float64
```

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

```
In [9]: a = np.zeros(3, dtype=int)

In [10]: type(a[0])
Out[10]: numpy.int64
```

**Shape and Dimension** Consider the following assignment

```
In [11]: z = np.zeros(10)
```

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

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

```
In [12]: z.shape
Out[12]: (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

```
In [13]: z.shape = (10, 1)
```

```
In [14]: z
Out[14]:
array([[ 0.],
       [ 0.],
```

```
[ 0.],
[ 0.],
[ 0.],
[ 0.],
[ 0.],
[ 0.],
[ 0.],
[ 0.])
In [15]: z = np.zeros(4)
In [16]: z.shape = (2, 2)
In [17]: z
Out[17]:
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

```
In [18]: z = np.empty(3)
In [19]: z
Out[19]: 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`

```
In [20]: z = np.linspace(2, 4, 5) # From 2 to 4, with 5 elements
```

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

```
In [21]: z = np.identity(2)
In [22]: z
Out[22]:
array([[ 1.,  0.],
       [ 0.,  1.]])
```

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

```
In [23]: z = np.array([10, 20])          # ndarray from Python list

In [24]: z
Out[24]: array([10, 20])

In [25]: type(z)
Out[25]: numpy.ndarray

In [26]: z = np.array((10, 20), dtype=float)    # Here 'float' is equivalent to 'np.float64'

In [27]: z
Out[27]: array([ 10.,  20.])

In [28]: z = np.array([[1, 2], [3, 4]])        # 2D array from a list of lists

In [29]: z
Out[29]:
array([[1, 2],
       [3, 4]])
```

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

```
In [11]: na = np.linspace(10, 20, 2)

In [12]: na is np.asarray(na)      # Does not copy NumPy arrays
Out[12]: True

In [13]: na is np.array(na)       # Does make a new copy --- perhaps unnecessarily
Out[13]: 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:

```
In [30]: z = np.linspace(1, 2, 5)

In [31]: z
Out[31]: array([ 1. ,  1.25,  1.5 ,  1.75,  2. ])

In [32]: z[0]
Out[32]: 1.0

In [33]: z[0:2] # Two elements, starting at element 0
Out[33]: array([ 1. ,  1.25])

In [34]: z[-1]
Out[34]: 2.0
```

For 2D arrays the index syntax is as follows:

```
In [35]: z = np.array([[1, 2], [3, 4]])

In [36]: z
Out[36]:
array([[1, 2],
       [3, 4]])

In [37]: z[0, 0]
Out[37]: 1

In [38]: z[0, 1]
Out[38]: 2
```

And so on

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

Columns and rows can be extracted as follows

```
In [39]: z[0,:]
Out[39]: array([1, 2])

In [40]: z[:,1]
Out[40]: array([2, 4])
```

NumPy arrays of integers can also be used to extract elements

```
In [41]: z = np.linspace(2, 4, 5)

In [42]: z
Out[42]: array([ 2. ,  2.5,  3. ,  3.5,  4. ])

In [43]: indices = np.array((0, 2, 3))

In [44]: z[indices]
Out[44]: array([ 2. ,  3. ,  3.5])
```

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

```
In [45]: z
Out[45]: array([ 2. ,  2.5,  3. ,  3.5,  4. ])

In [46]: d = np.array([0, 1, 1, 0, 0], dtype=bool)

In [47]: d
Out[47]: array([False,  True,  True, False, False], dtype=bool)

In [48]: z[d]
Out[48]: array([ 2.5,  3. ])
```

We'll see why this is useful below

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

```
In [49]: z = np.empty(3)

In [50]: z
Out[50]: array([-1.25236750e-041, 0.00000000e+000, 5.45693855e-313])

In [51]: z[:] = 42

In [52]: z
Out[52]: array([42., 42., 42.])
```

**Array Methods** Arrays have useful methods, all of which are carefully optimized

```
In [53]: A = np.array((4, 3, 2, 1))

In [54]: A
Out[54]: array([4, 3, 2, 1])

In [55]: A.sort()          # Sorts A in place

In [56]: A
Out[56]: array([1, 2, 3, 4])

In [57]: A.sum()           # Sum
Out[57]: 10

In [58]: A.mean()          # Mean
Out[58]: 2.5

In [59]: A.max()           # Max
Out[59]: 4

In [60]: A.argmax()        # Returns the index of the maximal element
Out[60]: 3

In [61]: A.cumsum()         # Cumulative sum of the elements of A
Out[61]: array([1, 3, 6, 10])

In [62]: A.cumprod()        # Cumulative product of the elements of A
Out[62]: array([1, 2, 6, 24])

In [63]: A.var()            # Variance
Out[63]: 1.25

In [64]: A.std()             # Standard deviation
Out[64]: 1.1180339887498949

In [65]: A.shape = (2, 2)

In [66]: A.T                # Equivalent to A.transpose()
Out[66]:
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  $\geq a$

```
In [67]: z = np.linspace(2, 4, 5)

In [68]: z
Out[68]: array([ 2.,  2.5,  3.,  3.5,  4.])

In [69]: z.searchsorted(2.2)
Out[69]: 1

In [70]: z.searchsorted(2.5)
Out[70]: 1

In [71]: z.searchsorted(2.6)
Out[71]: 2
```

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

```
In [72]: a = np.array((4, 3, 2, 1))

In [73]: np.sum(a)
Out[73]: 10

In [74]: np.mean(a)
Out[74]: 2.5
```

## Operations on Arrays

**Algebraic Operations** The algebraic operators `+`, `-`, `*`, `/` and `**` all act *elementwise* on arrays

```
In [75]: a = np.array([1, 2, 3, 4])

In [76]: b = np.array([5, 6, 7, 8])

In [77]: a + b
Out[77]: array([ 6,  8, 10, 12])

In [78]: a * b
Out[78]: array([ 5, 12, 21, 32])
```

We can add a scalar to each element as follows

```
In [79]: a + 10
Out[79]: array([11, 12, 13, 14])
```

Scalar multiplication is similar

```
In [81]: a = np.array([1, 2, 3, 4])
```

```
In [82]: a * 10
Out[82]: array([10, 20, 30, 40])
```

The two dimensional arrays follow the same general rules

```
In [86]: A = np.ones((2, 2))
```

```
In [87]: B = np.ones((2, 2))
```

```
In [88]: A + B
```

```
Out[88]:
```

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

```
In [89]: A + 10
```

```
Out[89]:
```

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

```
In [90]: A * B
```

```
Out[90]:
```

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

In particular,  $A * B$  is *not* the matrix product, it is an elementwise 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:

```
In [1]: import numpy as np
```

```
In [2]: A = np.ones((2, 2))
```

```
In [3]: B = np.ones((2, 2))
```

```
In [4]: A @ B
```

```
Out[4]:
```

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

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

We can also use `@` to take the inner product of two flat arrays

```
In [5]: A = np.array((1, 2))
```

```
In [6]: B = np.array((10, 20))
```

```
In [7]: A @ B
```

```
Out[7]: 50
```

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

```
In [11]: A = np.array(((1, 2), (3, 4)))
```

```
In [12]: A
```

```
Out[12]:  
array([[1, 2],  
       [3, 4]])
```

```
In [13]: A @ (0, 1)
```

```
Out[13]: 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:

```
In [21]: a = np.array([42, 44])
```

```
In [22]: a
```

```
Out[22]: array([42, 44])
```

```
In [23]: a[-1] = 0 # Change last element to 0
```

```
In [24]: a
```

```
Out[24]: array([42, 0])
```

Mutability leads to the following behavior, which some people find seem to find shocking

```
In [16]: a = np.random.randn(3)
```

```
In [17]: a
```

```
Out[17]: array([ 0.69695818, -0.05165053, -1.12617761])
```

```
In [18]: b = a
```

```
In [19]: b[0] = 0.0
```

```
In [20]: a
```

```
Out[20]: array([ 0. , -0.05165053, -1.12617761])
```

We have changed a by changing b

This works because of the Python assignment model described in detail earlier in the course

The name b is bound to a and becomes just another reference to the array

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`

```
In [14]: a = np.random.randn(3)

In [15]: a
Out[15]: array([ 0.67357176, -0.16532174,  0.36539759])

In [16]: b = np.empty_like(a)  # empty array with same shape as a

In [17]: np.copyto(b, a)    # copy to b from a

In [18]: b
Out[18]: array([ 0.67357176, -0.16532174,  0.36539759])
```

Now `b` is an independent copy (often called a *deep copy*)

```
In [19]: b[:] = 1

In [20]: b
Out[20]: array([ 1.,  1.,  1.])

In [21]: a
Out[21]: 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 *elementwise* on arrays

```
In [110]: z = np.array([1, 2, 3])

In [111]: np.sin(z)
Out[111]: array([ 0.84147098,  0.90929743,  0.14112001])
```

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

```
for i in range(n):
    y[i] = np.sin(z[i])
```

Because they act elementwise 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 elementwise, and combining these with the ufuncs gives a very large set of fast elementwise functions

```
In [112]: z
Out[112]: array([1, 2, 3])

In [113]: (1 / np.sqrt(2 * np.pi)) * np.exp(- 0.5 * z**2)
Out[113]: array([ 0.24197072,  0.05399097,  0.00443185])
```

Not all user defined functions will act elementwise

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:

```
In [114]: import numpy as np

In [115]: x = np.random.randn(4)

In [116]: x
Out[116]: array([-0.25521782,  0.38285891, -0.98037787, -0.083662  ])

In [117]: np.where(x > 0, 1, 0)  # Insert 1 if x > 0 true, otherwise 0
Out[117]: array([0, 1, 0, 0])
```

You can also use `np.vectorize` to vectorize a given function

```
In [118]: def f(x): return 1 if x > 0 else 0

In [119]: f = np.vectorize(f)

In [120]: f(x)                      # Passing same vector x as previous example
Out[120]: 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 elementwise

```
In [97]: z = np.array([2, 3])

In [98]: y = np.array([2, 3])

In [99]: z == y
Out[99]: array([ True,  True], dtype=bool)

In [100]: y[0] = 5

In [101]: z == y
Out[101]: array([False,  True], dtype=bool)
```

```
In [102]: z != y
Out[102]: array([ True, False], dtype=bool)
```

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

We can also do comparisons against scalars

```
In [103]: z = np.linspace(0, 10, 5)

In [104]: z
Out[104]: array([ 0., 2.5, 5., 7.5, 10.])

In [105]: z > 3
Out[105]: array([False, False, True, True, True], dtype=bool)
```

This is particularly useful for *conditional extraction*

```
In [106]: b = z > 3

In [107]: b
Out[107]: array([False, False, True, True, True], dtype=bool)

In [108]: z[b]
Out[108]: array([ 5., 7.5, 10.])
```

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

```
In [109]: z[z > 3]
Out[109]: 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`

```
In [134]: z = np.random.randn(10000) # Generate standard normals

In [135]: y = np.random.binomial(10, 0.5, size=1000) # 1,000 draws from Bin(10, 0.5)

In [136]: y.mean()
Out[136]: 5.036999999999999
```

Another commonly used subpackage is `np.linalg`

```
In [131]: A = np.array([[1, 2], [3, 4]])

In [132]: np.linalg.det(A) # Compute the determinant
Out[132]: -2.0000000000000004

In [133]: np.linalg.inv(A) # Compute the inverse
Out[133]:
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

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots a_Nx^N = \sum_{n=0}^N a_nx^n \quad (1.8)$$

*Earlier*, you wrote a simple function `p(x, coeff)` to evaluate (1.8) 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  $I_0, I_1, \dots, I_{n-1}$  such that the length of  $I_i$  is  $q_i$
- Draw a uniform random variable `U` on  $[0, 1]$  and return the  $i$  such that  $U \in I_i$

The probability of drawing  $i$  is the length of  $I_i$ , which is equal to  $q_i$

We can implement the algorithm as follows

```
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 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

[Solution notebook](#)

# Matplotlib

## Contents

- *Matplotlib*
  - *Overview*
  - *A Simple 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.

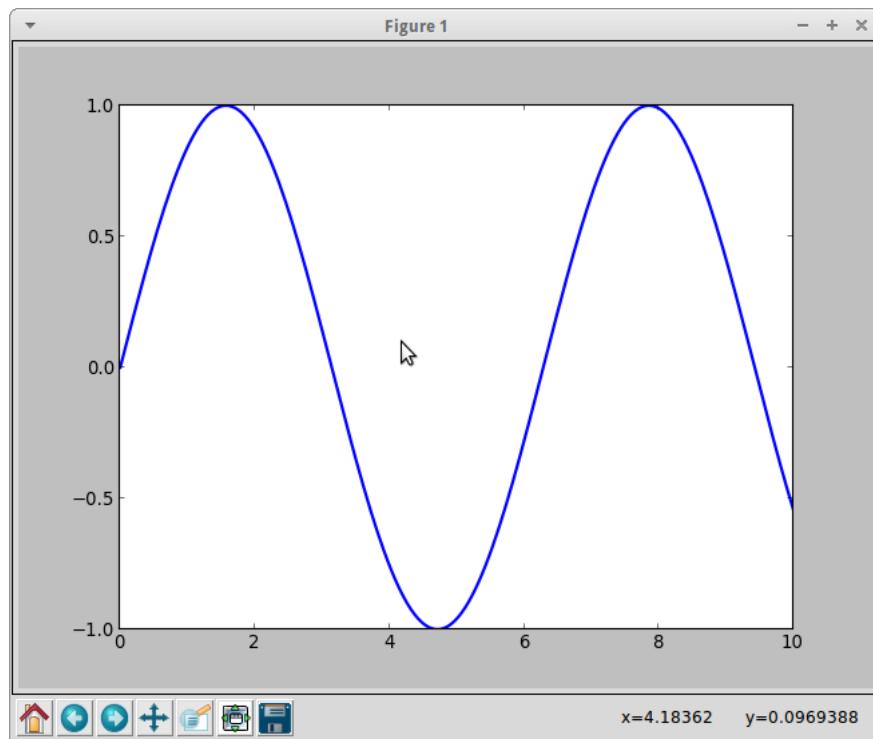
### A Simple API

Matplotlib is very easy to get started with, thanks to its simple MATLAB-style API (Application Programming Interface)

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

An 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.

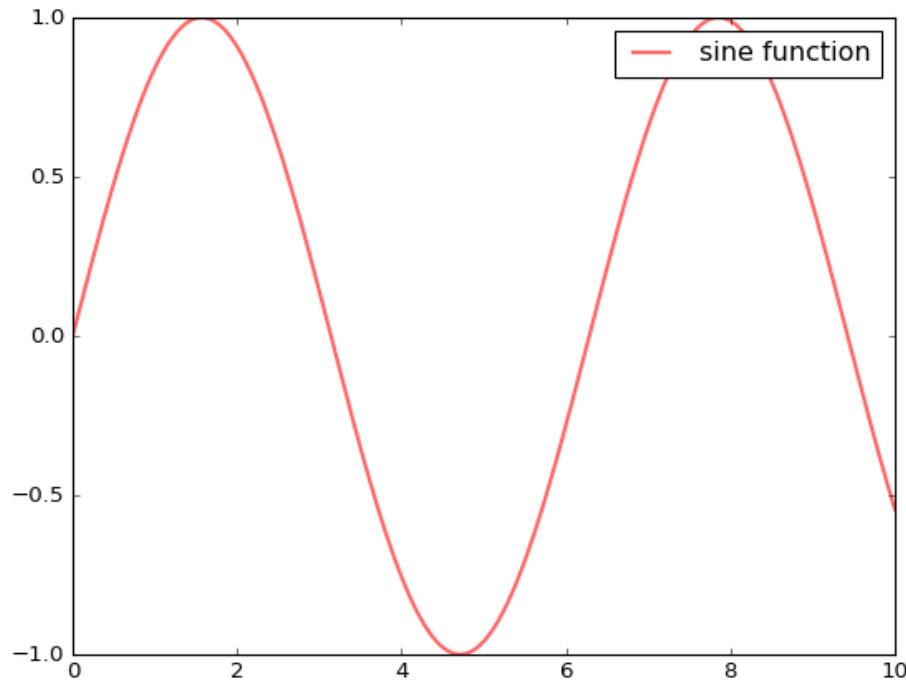
Incidentally, regarding the above lines of code,

- 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)
```

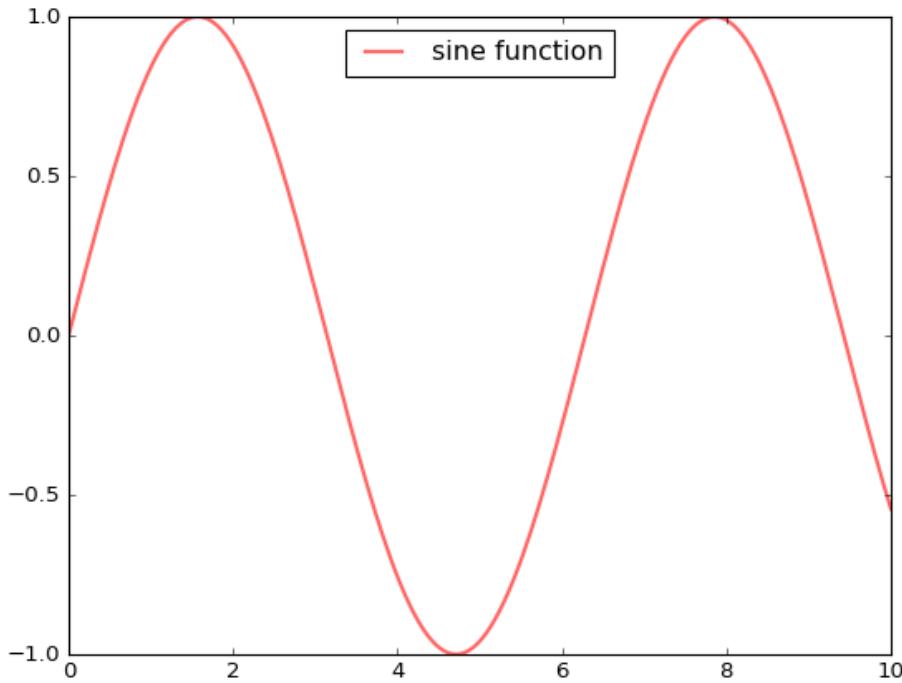
```
ax.legend()  
plt.show()
```



We've also used `alpha` to make the line slightly transparent—which makes it look smoother

Unfortunately the legend is obscuring the line

This can be fixed by replacing `ax.legend()` with `ax.legend(loc='upper center')`

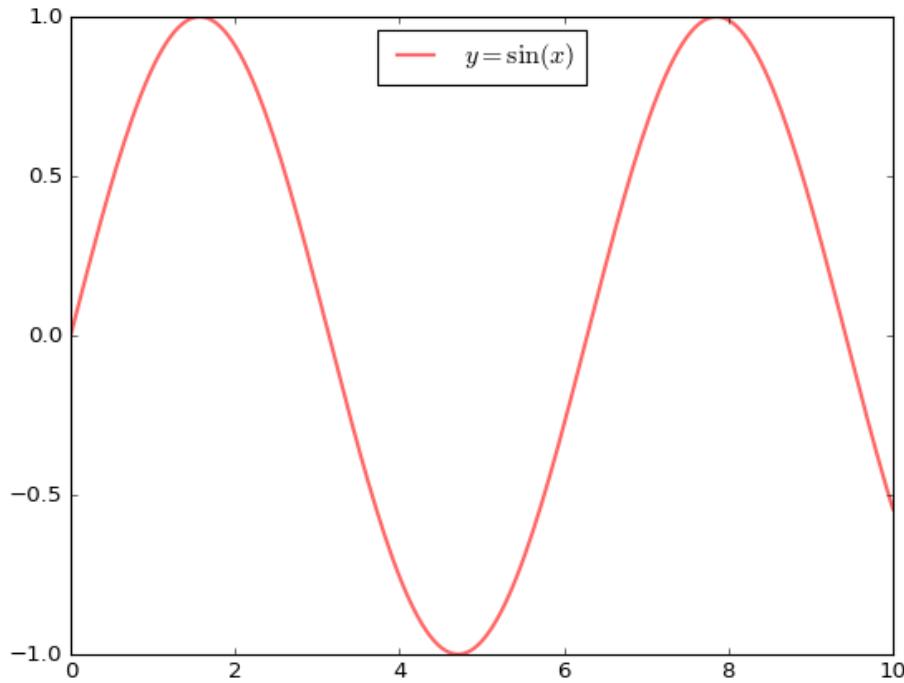


If everything is properly configured, then adding LaTeX is trivial

```
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

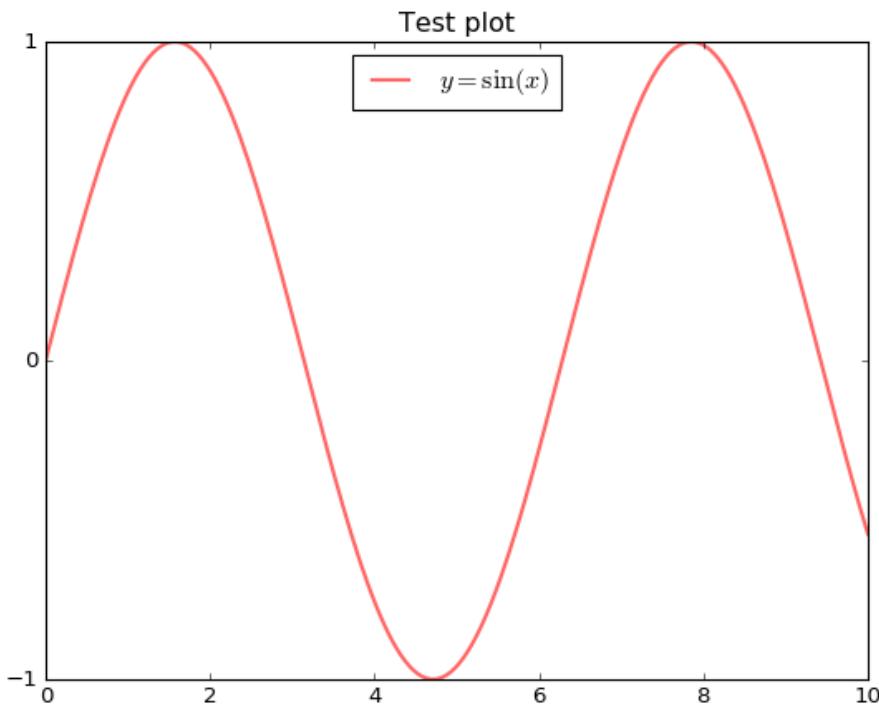
The figure now looks as follows



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

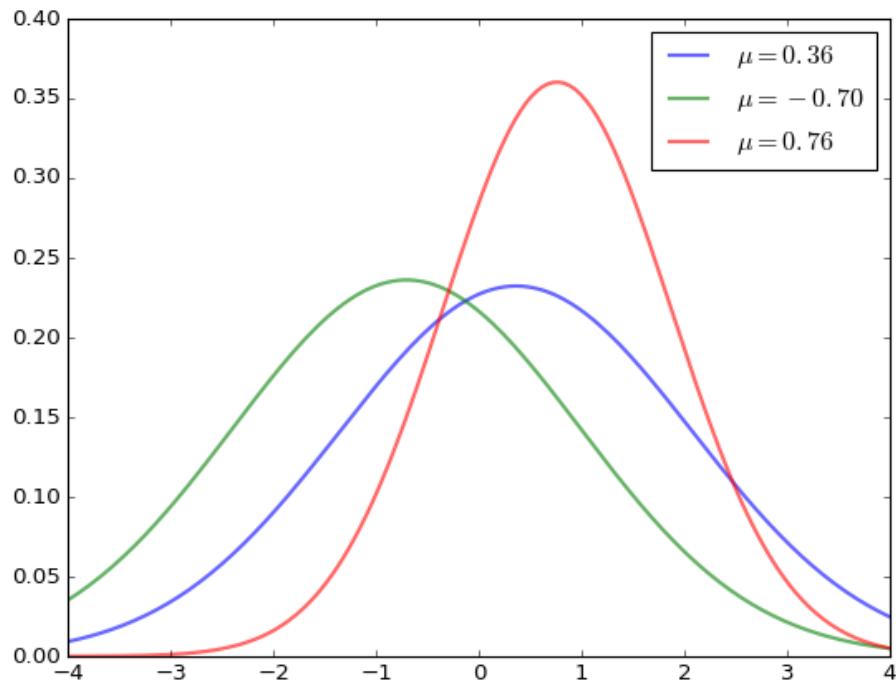
We mention just a few

**Multiple Plots on One Axis** It's straightforward to generate multiple 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'$\mu = {:.2f}$'.format(m)
    ax.plot(x, y, linewidth=2, alpha=0.6, label=current_label)
ax.legend()
plt.show()
```

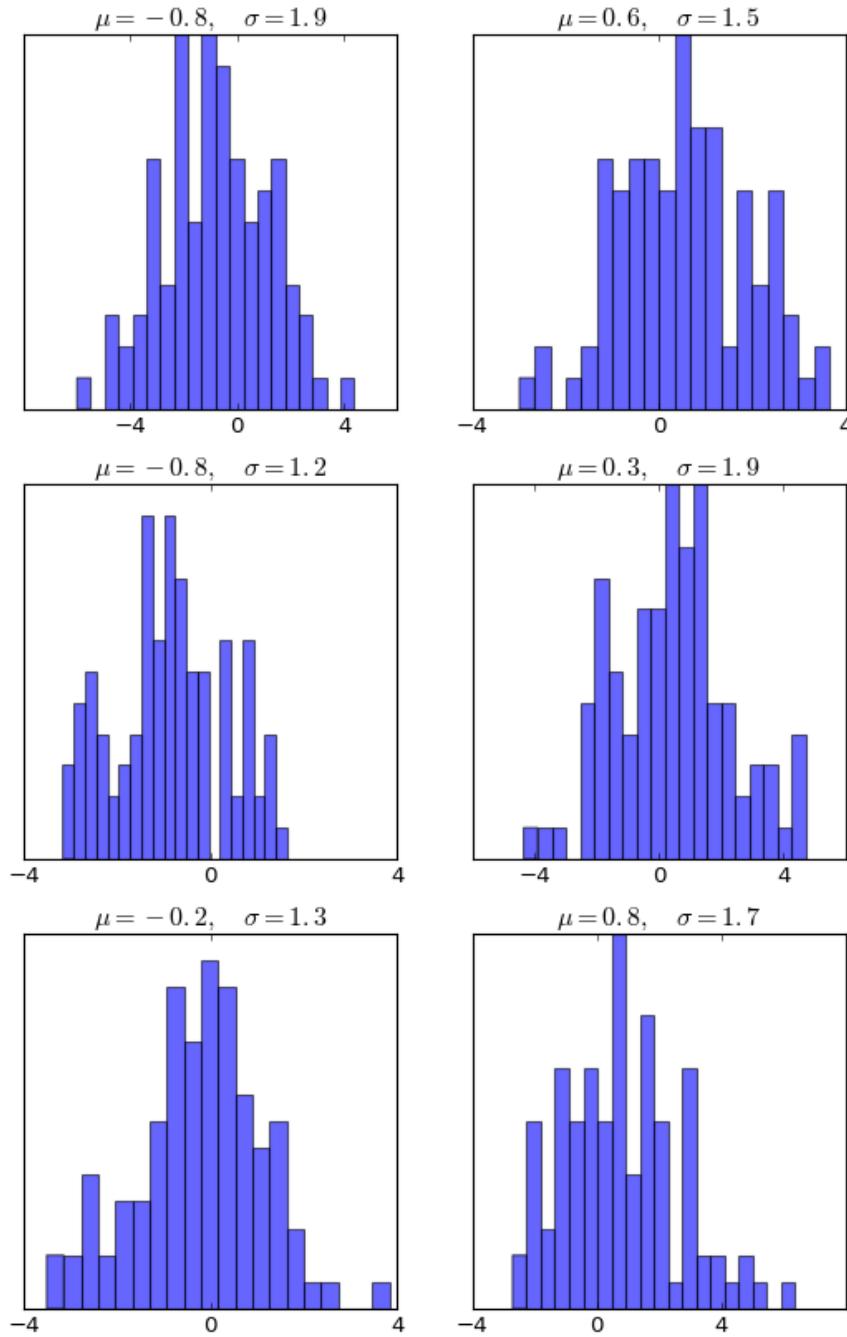


**Multiple Subplots** Sometimes we want multiple subplots in one figure

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'$\mu = {:.1f}, \sigma = {:.1f}$'.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

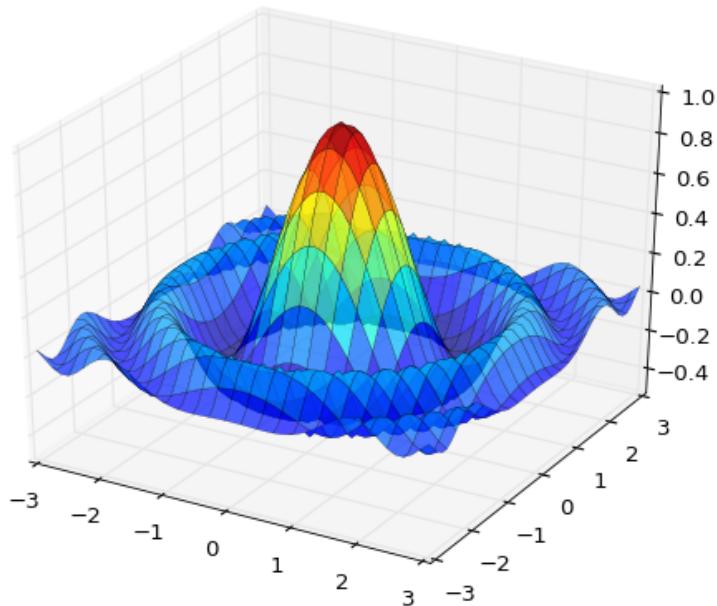


In fact the preceding figure was generated by the code above preceded by the following three lines

```
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

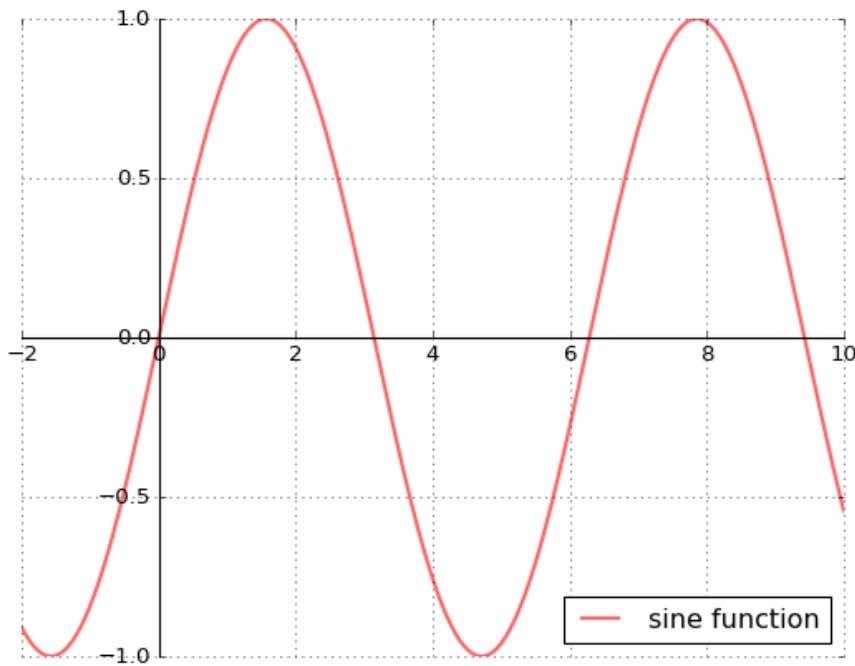
def subplots():
    "Custom subplots with axes through the origin"
    fig, ax = plt.subplots()

    # Set the axes through the origin
    for spine in ['left', 'bottom']:
        ax.spines[spine].set_position('zero')
    for spine in ['right', 'top']:
        ax.spines[spine].set_color('none')

    ax.grid()
    return fig, ax

fig, ax = subplots()  # Call the local version, not plt.subplots()
x = np.linspace(-2, 10, 200)
y = np.sin(x)
ax.plot(x, y, 'r-', linewidth=2, label='sine function', alpha=0.6)
ax.legend(loc='lower right')
plt.show()
```

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



The custom `subplots` function

1. calls the standard `plt.subplots` function internally to generate the `fig`, `ax` pair,
2. makes the desired customizations to `ax`, and
3. passes the `fig`, `ax` pair back to the calling code

#### 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__ += _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

```
In [1]: import numpy as np

In [2]: 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

```
In [1]: import numpy as np

In [2]: np.random.beta(5, 5, size=3)
Out[2]: array([ 0.6167565 ,  0.67994589,  0.32346476])
```

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

$$f(x; a, b) = \frac{x^{(a-1)}(1-x)^{(b-1)}}{\int_0^1 u^{(a-1)}u^{(b-1)}du} \quad (0 \leq x \leq 1) \quad (1.9)$$

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

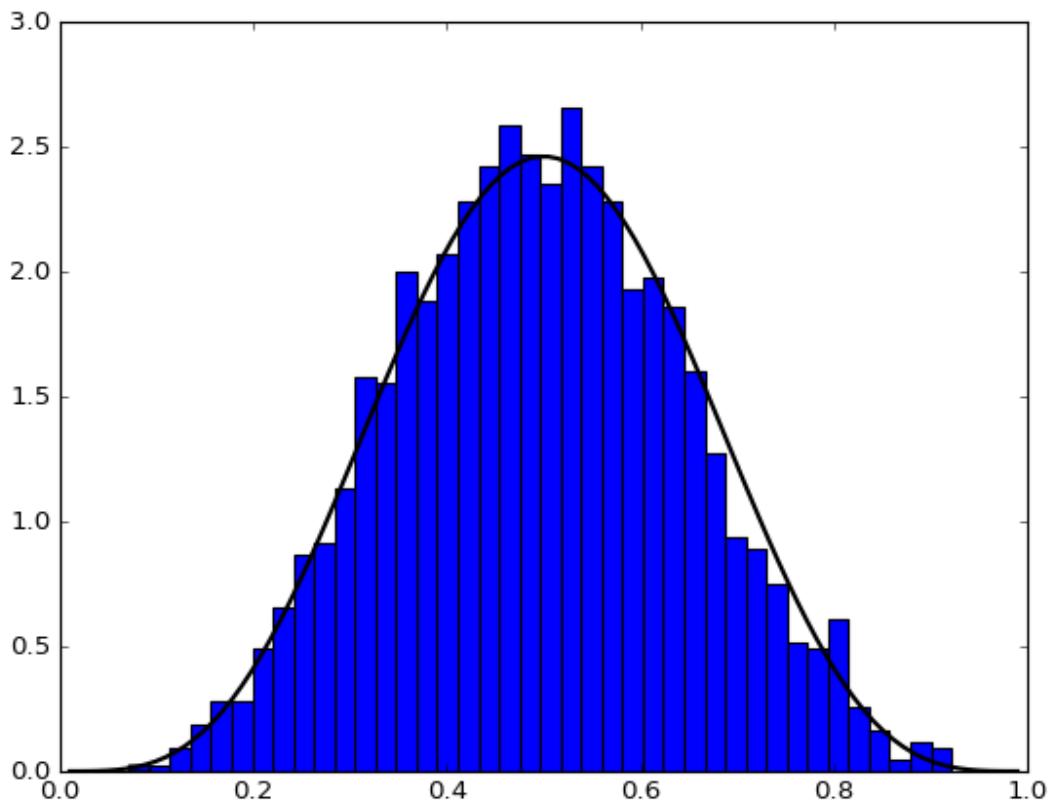
Here's an example of usage

```
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)
fig.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

```
In [14]: q.cdf(0.4)      # Cumulative distribution function
Out[14]: 0.2665676800000002

In [15]: q.pdf(0.4)      # Density function
Out[15]: 2.0901888000000004

In [16]: q.ppf(0.8)      # Quantile (inverse cdf) function
Out[16]: 0.63391348346427079

In [17]: q.mean()
Out[17]: 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`:

```
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)
fig.show()
```

**Other Goodies in `scipy.stats`** There are a variety statistical functions in `scipy.stats`

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

```
In [19]: from scipy.stats import linregress

In [20]: x = np.random.randn(200)
```

```
In [21]: y = 2 * x + 0.1 * np.random.randn(200)
In [22]: gradient, intercept, r_value, p_value, std_err = linregress(x, y)
In [23]: gradient, intercept
Out[23]: (1.9962554379482236, 0.008172822032671799)
```

To see the full list, consult the documentation

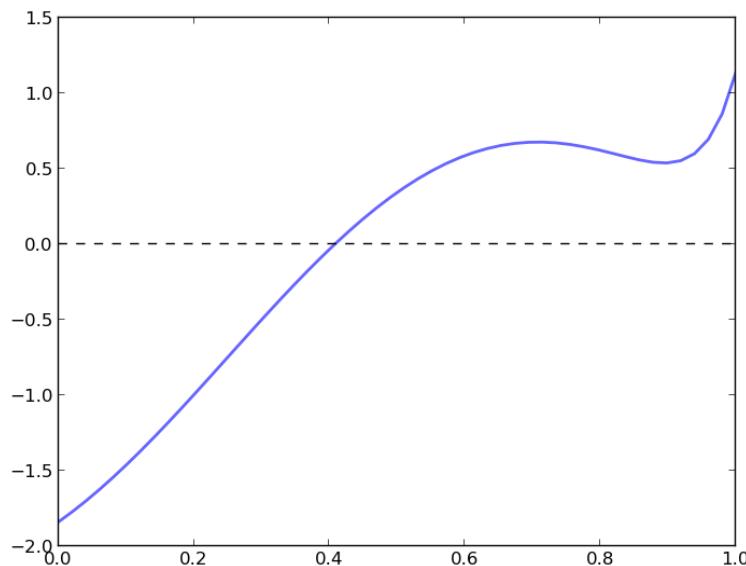
### Roots and Fixed Points

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

$$f(x) = \sin(4(x - 1/4)) + x + x^{20} - 1 \quad (1.10)$$

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 its own bisection function, which we now test using the function  $f$  defined in (1.10)

```
In [24]: from scipy.optimize import bisect

In [25]: f = lambda x: np.sin(4 * (x - 0.25)) + x + x**20 - 1

In [26]: bisect(f, 0, 1)
Out[26]: 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.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

```
In [27]: from scipy.optimize import newton

In [28]: newton(f, 0.2)    # Start the search at initial condition x = 0.2
Out[28]: 0.40829350427935679

In [29]: newton(f, 0.7)    # Start the search at x = 0.7 instead
Out[29]: 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

```
In [32]: timeit bisect(f, 0, 1)
1000 loops, best of 3: 261 us per loop

In [33]: 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 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

```
In [35]: brentq(f, 0, 1)
Out[35]: 0.40829350427936706

In [36]: 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

```
In [1]: from scipy.optimize import fixed_point
In [2]: fixed_point(lambda x: x**2, 10.0) # 10.0 is an initial guess
Out[2]: 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`

```
In [9]: from scipy.optimize import fminbound
In [10]: fminbound(lambda x: x**2, -1, 2) # Search in [-1, 2]
Out[10]: 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`

```
In [13]: from scipy.integrate import quad
In [14]: integral, error = quad(lambda x: x**2, 0, 1)
In [15]: integral
Out[15]: 0.3333333333333337
```

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

[Solution notebook](#)

# 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

Just as NumPy provides the basic array type plus core array operations, pandas defines some fundamental structures for working with data and endows them with methods that form the first steps of data analysis

The most important data type defined by pandas is a `DataFrame`, which is an object for storing related columns of data

In this sense, you can think of a `DataFrame` as analogous to a (highly optimized) Excel spreadsheet, or as a structure for storing the `X` matrix in a linear regression

In the same way that NumPy specializes in basic array operations and leaves the rest of scientific tool development to other packages (e.g., SciPy, Matplotlib), pandas focuses on the fundamental data types and their methods, leaving other packages to add more sophisticated statistical functionality

The strengths of pandas lie in

- reading in data
- manipulating rows and columns
- adjusting indices
- working with dates and time series
- sorting, grouping, re-ordering and general data munging <sup>1</sup>
- dealing with missing values, etc., etc.

This lecture will provide a basic introduction

Throughout the lecture we will assume that the following imports have taken place

```
In [1]: import pandas as pd
In [2]: import numpy as np
```

## Series

Perhaps the two most important data types defined by pandas are the `DataFrame` and `Series` types

You can think of a `Series` as a “column” of data, such as a collection of observations on a single variable

```
In [4]: s = pd.Series(np.random.randn(4), name='daily returns')
In [5]: s
Out[5]:
0    0.430271
1    0.617328
2   -0.265421
3   -0.836113
Name: daily returns
```

Here you can imagine the indices 0, 1, 2, 3 as indexing four listed companies, and the values being daily returns on their shares

Pandas `Series` are built on top of NumPy arrays, and support many similar operations

```
In [6]: s * 100
Out[6]:
0    43.027108
1    61.732829
2   -26.542104
3   -83.611339
Name: daily returns

In [7]: np.abs(s)
Out[7]:
0    0.430271
```

---

<sup>1</sup> Wikipedia defines munging as cleaning data from one raw form into a structured, purged one.

```

1    0.617328
2    0.265421
3    0.836113
Name: daily returns

```

But `Series` provide more than NumPy arrays

Not only do they have some additional (statistically oriented) methods

```

In [8]: s.describe()
Out[8]:
count    4.000000
mean     -0.013484
std      0.667092
min     -0.836113
25%     -0.408094
50%      0.082425
75%      0.477035
max      0.617328

```

But their indices are more flexible

```

In [9]: s.index = ['AMZN', 'AAPL', 'MSFT', 'GOOG']

In [10]: s
Out[10]:
AMZN    0.430271
AAPL    0.617328
MSFT   -0.265421
GOOG   -0.836113
Name: daily returns

```

Viewed in this way, `Series` are like fast, efficient Python dictionaries (with the restriction that the items in the dictionary all have the same type—in this case, floats)

In fact you can use much of the same syntax as Python dictionaries

```

In [11]: s['AMZN']
Out[11]: 0.43027108469945924

In [12]: s['AMZN'] = 0

In [13]: s
Out[13]:
AMZN    0.000000
AAPL    0.617328
MSFT   -0.265421
GOOG   -0.836113
Name: daily returns

In [14]: 'AAPL' in s
Out[14]: True

```

## DataFrames

As mentioned above a DataFrame is somewhat like a spreadsheet, or a structure for storing the data matrix in a regression

While a Series is one individual column of data, a DataFrame is all the columns

Let's look at an example, reading in data from the CSV file pandas/test\_pwt.csv in the applications repository

Here's the contents of test\_pwt.csv, which is a small excerpt from the Penn World Tables

```
"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"
```

Here we're in IPython, so we have access to shell commands such as ls, as well as the usual Python commands

```
In [15]: ls data/test_pwt* # List all files starting with 'test_pwt' -- check CSV file is in present working directory
test_pwt.csv
```

Now let's read the data in using pandas' read\_csv function

```
In [28]: df = pd.read_csv('data/test_pwt.csv')

In [29]: type(df)
Out[29]: pandas.core.frame.DataFrame

In [30]: df
Out[30]:
   country country isocode  year      POP      XRAT      tcgdp      cc      cg
0    Argentina        ARG  2000  37335.653  0.999500  295072.218690    0  75.716805  5.5788042896
1    Australia        AUS  2000  19053.186  1.724830  541804.652100    1  67.759026  6.7200975332
2      India         IND  2000 1006300.297  44.941600 1728144.374800    2  64.575551 14.072205773
3     Israel         ISR  2000   6114.570  4.077330 129253.894230    3  64.436451 10.266688415
4    Malawi         MWI  2000  11801.505  59.543808  5026.221784    4  74.707624 11.658954494
5  South Africa       ZAF  2000  45064.098  6.939830 227242.369490    5  72.718710  5.7265463933
6   United States       USA  2000 282171.957  1.000000 9898700.000000    6  72.347054  6.0324539789
7    Uruguay         URY  2000   3219.793 12.099592  25255.961693    7  78.978740  5.108067988
```

We can select particular rows using standard Python array slicing notation

```
In [13]: df[2:5]
Out[13]:
   country country isocode  year      POP      XRAT      tcgdp      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

```
In [14]: df[['country', 'tcgdp']]
Out[14]:
      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 a mix of both we can use the `ix` attribute

```
In [21]: df.ix[2:5, ['country', 'tcgdp']]
Out[21]:
      country          tcgdp
2      India  1728144.374800
3    Israel  129253.894230
4    Malawi  5026.221784
5  South Africa  227242.369490
```

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 as follows

```
In [31]: keep = ['country', 'POP', 'tcgdp']

In [32]: df = df[keep].copy()

In [33]: df
Out[33]:
      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, first let's pull out the `country` column using the `pop` method

```
In [34]: countries = df.pop('country')

In [35]: type(countries)
```

```
Out[35]: pandas.core.series.Series
```

```
In [36]: countries
```

```
Out[36]:
```

0	Argentina
1	Australia
2	India
3	Israel
4	Malawi
5	South Africa
6	United States
7	Uruguay

```
Name: country
```

```
In [37]: df
```

```
Out[37]:
```

	POP	tcgdp
0	37335.653	295072.218690
1	19053.186	541804.652100
2	1006300.297	1728144.374800
3	6114.570	129253.894230
4	11801.505	5026.221784
5	45064.098	227242.369490
6	282171.957	9898700.000000
7	3219.793	25255.961693

```
In [38]: df.index = countries
```

```
In [39]: df
```

```
Out[39]:
```

country	POP	tcgdp
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

```
In [40]: df.columns = 'population', 'total GDP'
```

```
In [41]: df
```

```
Out[41]:
```

country	population	total GDP
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

```
In [66]: df['population'] = df['population'] * 1e3
```

```
In [67]: df
```

```
Out[67]:
```

country	population	total GDP
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

```
In [74]: df['GDP percap'] = df['total GDP'] * 1e6 / df['population']
```

```
In [75]: df
```

```
Out[75]:
```

country	population	total GDP	GDP percap
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

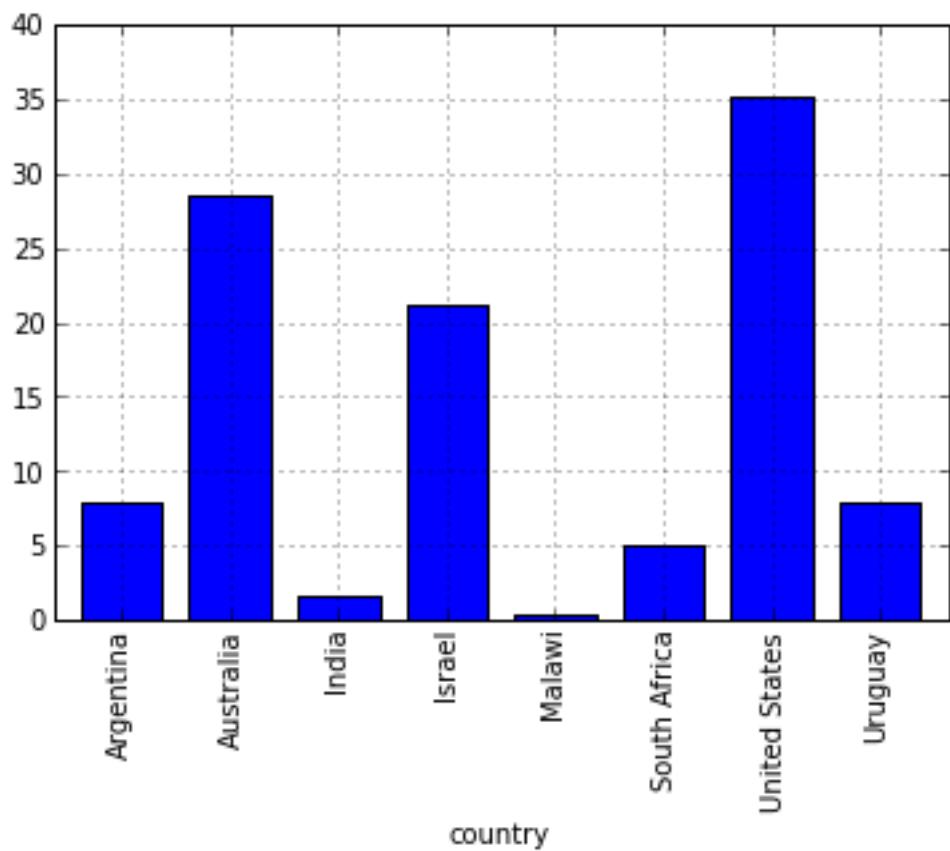
```
In [76]: df['GDP percap'].plot(kind='bar')
```

```
Out[76]: <matplotlib.axes.AxesSubplot at 0x2f22ed0>
```

```
In [77]: import matplotlib.pyplot as plt
```

```
In [78]: 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

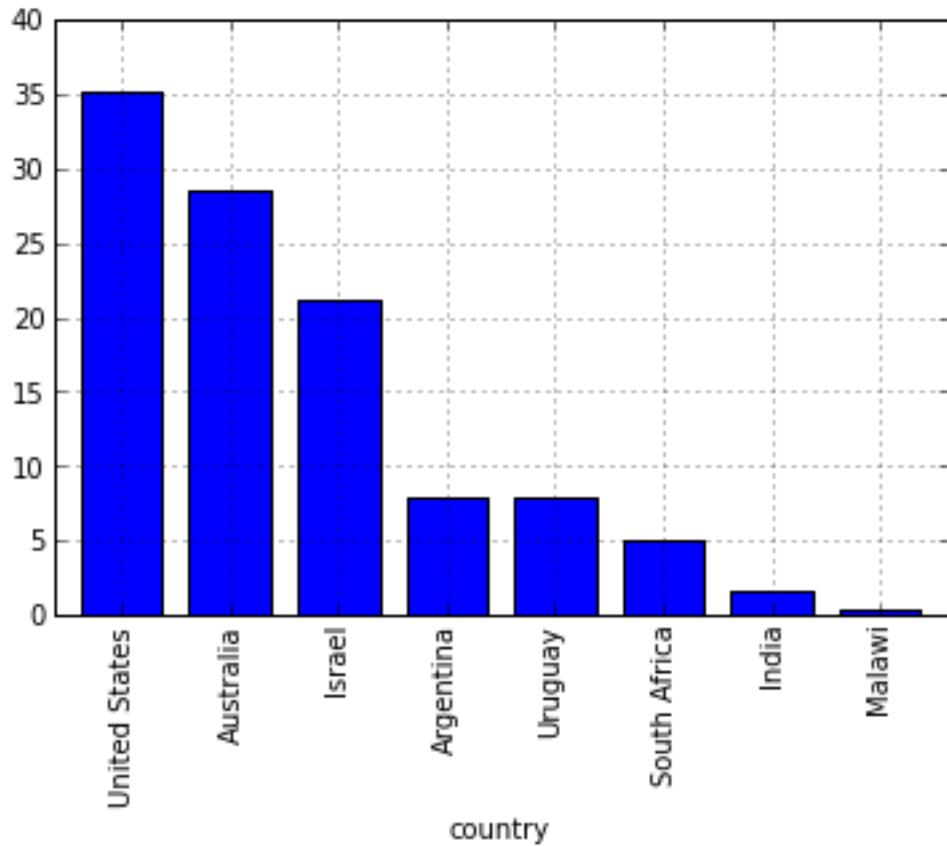
```
In [83]: df = df.sort_values(by='GDP percap', ascending=False)
```

```
In [84]: df
```

```
Out[84]:
```

country	population	total GDP	GDP percap
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

pandas makes it straightforward to query several common Internet databases programmatically

One particularly important one 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 rate can be downloaded directly by entering this URL into your browser

```
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 urllib.request** One option is to use **urllib.request**, a standard Python library for requesting data over the Internet

To begin, try the following code on your computer

```
In [36]: import urllib.request
```

```
In [37]: source = urllib.request.urlopen('http://research.stlouisfed.org/fred2/series/UNRATE/downloaddat
```

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 (for example, log in to Wakari)
- 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 `urllib.request.urlopen('http://research.stlouisfed.org/fred2/series/UNRATE/downloaddata/UNRATE.csv')`

```
In [56]: url = 'http://research.stlouisfed.org/fred2/series/UNRATE/downloaddata/UNRATE.csv'
```

```
In [57]: source = urllib.request.urlopen(url).read().decode('utf-8').split("\n")
```

```
In [58]: source[0]
```

```
Out[58]: 'DATE,VALUE\r\n'
```

```
In [59]: source[1]
```

```
Out[59]: '1948-01-01,3.4\r\n'
```

```
In [60]: source[2]
Out[60]: '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

```
In [69]: source = urllib.request.urlopen(url)

In [70]: data = pd.read_csv(source, index_col=0, parse_dates=True, header=None)
```

The data has been read into a pandas DataFrame called `data` that we can now manipulate in the usual way

```
In [71]: type(data)
Out[71]: pandas.core.frame.DataFrame

In [72]: data.head()  # A useful method to get a quick look at a data frame
Out[72]:
          1
0
DATE      VALUE
1948-01-01    3.4
1948-02-01    3.8
1948-03-01    4.0
1948-04-01    3.9

In [73]: data.describe()  # Your output might differ slightly
Out[73]:
          1
count    786
unique   81
top      5.4
freq     31
```

**Accessing Data with pandas** Although it is worth understanding the low level procedures, for the present case pandas can take care of all these messy details

(pandas puts a simple API (Application Programming Interface) on top of the kind of low level function calls we've just covered)

---

**Note:** You may need to install the `pandas_datareader` package using conda: `conda install pandas-datareader`

---

For example, we can obtain the same unemployment data for the period 2006–2012 inclusive as follows

```
In [77]: from pandas_datareader import data,wb

In [78]: import datetime as dt  # Standard Python date / time library
```

```
In [79]: start, end = dt.datetime(2006, 1, 1), dt.datetime(2012, 12, 31)

In [80]: data = data.DataReader('UNRATE', 'fred', start, end)

In [81]: type(data)
Out[81]: pandas.core.frame.DataFrame

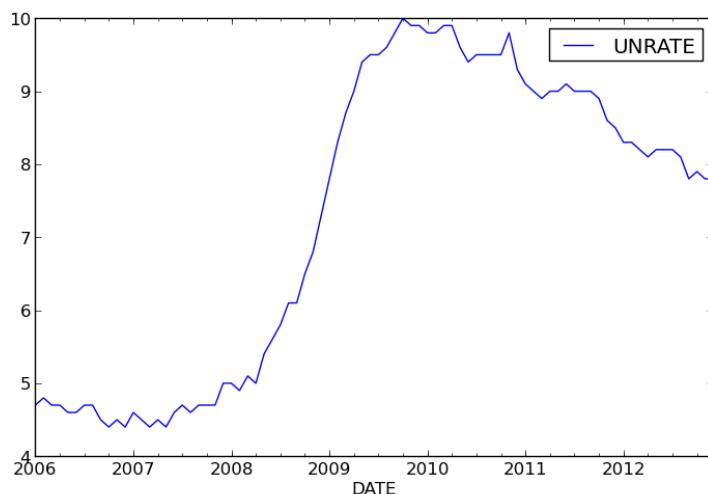
In [82]: data.plot()
Out[82]: <matplotlib.axes.AxesSubplot at 0xcf79390>

In [83]: import matplotlib.pyplot as plt

In [84]: plt.show()
```

(If you're working in the IPython notebook, the last two lines can probably be omitted)

The resulting figure looks as follows



**Data from the World Bank** 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 we find data on government debt as a ratio to GDP:  
<http://data.worldbank.org/indicator/GC.DOD.TOTL.GD.ZS/countries>

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, parses the data from Excel file to pandas DataFrame, and plots time series for France, Germany, the US and Australia

```
import sys
import matplotlib.pyplot as plt
```

```

from pandas.io.excel import ExcelFile

if sys.version_info[0] == 2:
    from urllib import urlretrieve
elif sys.version_info[0] == 3:
    from urllib.request import urlretrieve

# == 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"
urlretrieve(wb_data_query, "gd.xls")

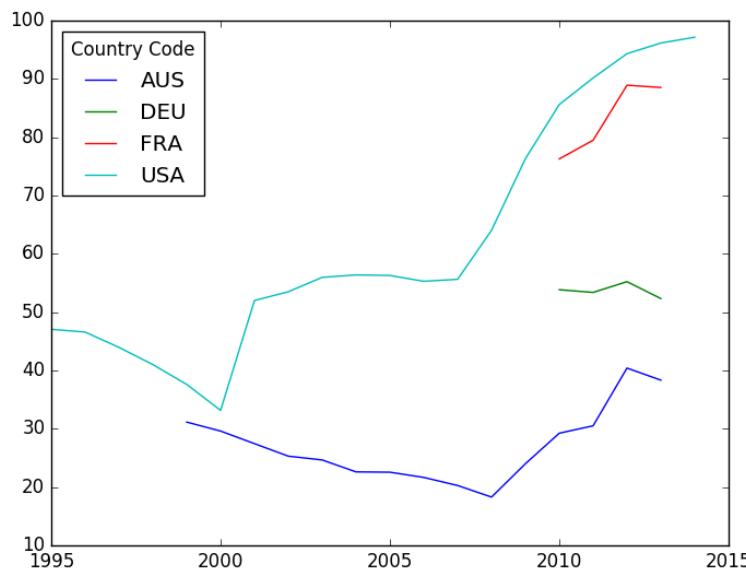
# == Parse data into a DataFrame ==
govt_debt_xls = ExcelFile('gd.xls')
govt_debt = govt_debt_xls.parse('Data', index_col=1, na_values=['NA'], skiprows=3)

# == 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` from the [applications repository](#))

The figure it produces looks as follows



(Missing line segments indicate missing data values)

Actually pandas includes high-level functions for downloading World Bank data

For example, see [http://pandas-docs.github.io/pandas-docs-travis/remote\\_data.html?highlight=world%20bank%20data-wb](http://pandas-docs.github.io/pandas-docs-travis/remote_data.html?highlight=world%20bank%20data-wb)

### Exercises

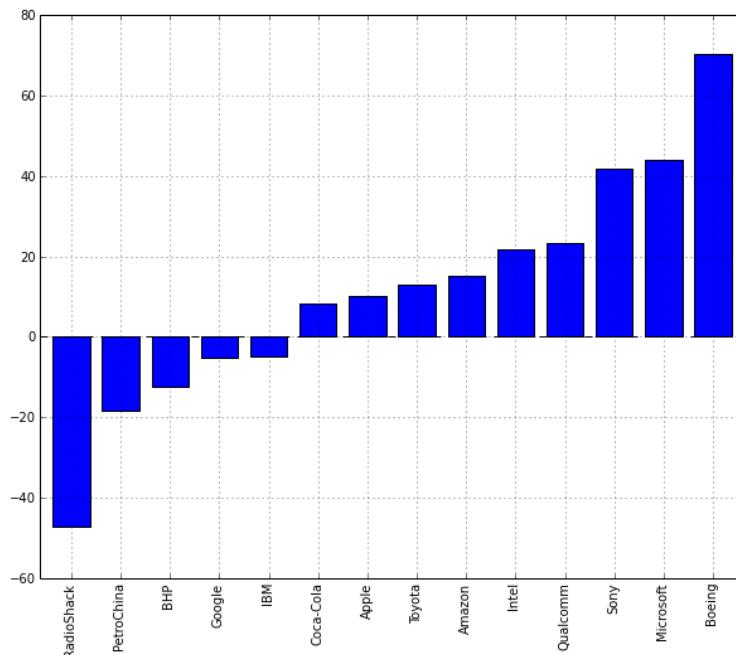
**Exercise 1** Write a program to calculate the percentage price change since the start of the year for the following shares

```
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'}
```

Use pandas to download the data from Yahoo Finance

Hint: Try replacing `data = web.DataReader('UNRATE', 'fred', start, end)` with `data = web.DataReader('AAPL', 'yahoo', start, end)` in the code above

Plot the result as a bar graph, such as this one (of course actual results will vary)



### Solutions

[Solution notebook](#)

## IPython Tips and Tricks

### Contents

- *IPython Tips and Tricks*
  - *Overview*
  - *IPython Magics*
  - *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

IPython is an enhanced Python command interface oriented towards scientific workflow

We've already mentioned the IPython shell (see for example in *this discussion*)

Moreover, when you run Python code in Jupyter, the content of each cell is sent to IPython to interpret

Here we briefly review some more of IPython's features

We will work in the IPython shell, but almost all of the following applies to the notebook too

### IPython Magics

**Line Magics** As we've seen, any Python command can be typed into an IPython shell

```
In [1]: 'foo' * 2  
Out[1]: 'foofoo'
```

But IPython understands more than just Python statements

For example, file `foo.py` in the present working directory can be executed using `run`

```
In [2]: run foo.py # Try %run instead of run if this doesn't work
```

Here `run` is not a Python command

Rather it is an IPython magic

IPython magics are divided into

1. Line magics, which precede and act on a single line
2. Cell magics, which act on a cell in a Jupyter notebook

In general,

- line magics need to be prefixed by %
  - for example, %run foo.py runs file foo.py
- cell magics need to be prefixed by %%
  - for example, %%latex means the contents of the cell will be interpreted as LaTeX

For line magics, you can toggle the need for % by running %automagic

Below we discuss some of the most important IPython magics

**Timing Code** For scientific work, we often want to know how long certain blocks of code take to run

For this purpose, IPython includes the `timeit` magic

Usage is straightforward — let's look at an example

In earlier exercises, we wrote two different functions to calculate the value of a polynomial

Let's put them in a file called `temp.py` as follows

```
## Filename: temp.py
import numpy as np

def p1(x, coef):
    return sum(a * x**i for i, a in enumerate(coef))

def p2(x, coef):
    X = np.empty(len(coef))
    X[0] = 1
    X[1:] = x
    y = np.cumprod(X)    # y = [1, x, x**2, ...]
    return np.dot(coef, y)
```

Note that `p1` uses pure Python, whereas `p2` uses NumPy arrays and should run faster

Here's how we can test this

```
In [1]: run temp.py

In [2]: p1(10, (1, 2))  # Let's make sure the function works OK
Out[2]: 21

In [3]: p2(10, (1, 2))  # Ditto
Out[3]: 21.0

In [4]: coef = np.random.randn(1000)

In [5]: timeit p1(0.9, coef)
1000 loops, best of 3: 1.15 ms per loop

In [6]: timeit p2(0.9, coef)
100000 loops, best of 3: 9.87 us per loop
```

For p1, average execution time was 1.15 milliseconds, while for p2 it was about 10 microseconds (i.e., millionths of a second) — two orders of magnitude faster

**Reloading Modules** Here is one very common Python gotcha and a nice solution provided by IPython

When we work with multiple files, changes in one file are not always visible in our program

To see this, suppose that you are working with files *useful\_functions.py* and *main\_program.py*

As the names suggest, the main program resides in *main\_program.py* but imports functions from *useful\_functions.py*

You might have noticed that if you make a change to *useful\_functions.py* and then re-run *main\_program.py*, the effect of that change isn't always apparent

Here's an example *useful\_functions.py* in the current directory

```
## Filename: useful_functions.py

def meaning_of_life():
    "Computes the meaning of life"
    return 42
```

Here is *main\_program.py*, which imports the former

```
## Filename: main_program.py

from useful_functions import meaning_of_life

x = meaning_of_life()
print("The meaning of life is: {}".format(x))
```

When we run *main\_program.py* we get the expected output

```
In [1]: run main_program.py
The meaning of life is: 42
```

Now suppose that we discover the meaning of life is actually 43

So we open up a text editor, and change the contents of *useful\_functions.py* to

```
## Filename: useful_functions.py

def meaning_of_life():
    "Computes the meaning of life"
    return 43
```

However, if we run *main\_program.py* again no change is visible

```
In [2]: run main_program.py
The meaning of life is: 42
```

The reason is that *useful\_functions.py* has been compiled to a byte code file, in preparation for sending its instructions to the Python virtual machine

The byte code file will be called `useful_functions.pyc`, and live in the same directory as `useful_functions.py`

Even though we've modified `useful_functions.py`, this change is not reflected in `useful_functions.pyc`

The nicest way to get your dependencies to recompile is to use IPython's *autoreload* extension

```
In [3]: %load_ext autoreload
```

```
In [4]: autoreload 2
```

Now save `useful_functions.py` again and try running it again

```
In [5]: run main_program.py
```

```
The meaning of life is: 43
```

The change is now reflected in our output

If you want this behavior to load automatically when you start IPython, add these lines to your `ipython_config.py` file

```
c.InteractiveShellApp.extensions = ['autoreload']
c.InteractiveShellApp.exec_lines = ['%autoreload 2']
```

Notes:

- Search your file system for this file
- If you can't find it, create one by typing `ipython profile create` in a terminal (not in IPython!)

Finally, if you prefer to do things manually, you can also `import` and then `reload` the modified module

```
In [3]: import useful_functions
```

```
In [4]: reload(useful_functions)
```

For any subsequent changes, you will only need `reload(useful_functions)`

## Debugging

Are you one of those programmers who fills their code with `print` statements when trying to debug their programs?

Hey, it's OK, we all used to do that

But today might be a good day to turn a new page, and start using a debugger

Debugging is a big topic, but it's actually very easy to learn the basics

The standard Python debugger is `pdb`

Here we use one called `ipdb` that plays well with the IPython shell

- If you don't have it, try `pip install ipdb` in a terminal

- But pdb will do the job fine too

Let's look at an example of when and how to use them

**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

```
In [2]: run temp.py
-----
AttributeError                                     Traceback (most recent call last)
/home/john/temp/temp.py  in <module>()
      8     plt.show()
      9
--> 10 plot_log() # Call the function, generate plot
/home/john/temp/temp.py  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

```
In [2]: ax
-----
```

```
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

First we run `temp.py` again, but this time we respond to the exception by typing `debug`

This will cause us to be dropped into the Python debugger at the point of execution just before the exception occurs

```
In [4]: run temp.py
-----
AttributeError Traceback (most recent call last)
/home/john/temp/temp.py in <module>()
      8     plt.show()
      9
---> 10 plot_log() # Call the function, generate plot

/home/john/temp/temp.py 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'

In [5]: debug
> /home/john/temp/temp.py(7)plot_log()
      6     x = np.linspace(1, 2, 10)
---> 7     ax.plot(x, np.log(x))
      8     plt.show()

ipdb>
```

We're now at the `ipdb>` prompt, at which 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 on line 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 `import ipdb; ipdb.set_trace()` inside the function code block:

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

def plot_log():
```

```

import ipdb; ipdb.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

```

In [3]: run temp.py
> /home/john/temp/temp.py(6)plot_log()
      5      import ipdb; ipdb.set_trace()
----> 6      fig, ax = plt.subplots()
      7      x = np.logspace(1, 2, 10)

ipdb> n
> /home/john/temp/temp.py(7)plot_log()
      6      fig, ax = plt.subplots()
----> 7      x = np.logspace(1, 2, 10)
      8      ax.plot(x, np.log(x))

ipdb> n
> /home/john/temp/temp.py(8)plot_log()
      7      x = np.logspace(1, 2, 10)
----> 8      ax.plot(x, np.log(x))
      9      plt.show()

ipdb> x
array([ 10.        ,   12.91549665,   16.68100537,   21.5443469 ,
       27.82559402,   35.93813664,   46.41588834,   59.94842503,
      77.42636827,   100.        ])

```

We used `n` twice to step forward through the code (one line at a time)

Then we printed the value of `x` to see what was happening with that variable

To exit from the debugger, use `q`

### Other Useful Magics

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](#)

## The Need for Speed

### Contents

- *The Need for Speed*
  - *Overview*
  - *Where are the Bottlenecks?*
  - *Vectorization*
  - *Numba*
  - *Cython*
  - *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.

One result is that, compared to low-level languages, Python is typically faster to write, less error prone and easier to debug

A 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 (no matter what your peers might tell you)

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 speeds 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

```
In [1]: a, b = 10, 10
In [2]: a + b
Out[2]: 20
```

Even for this simple operation, the Python interpreter has a fair bit of work to do

For example, in the statement  $a + b$ , the interpreter has to know which operation to invoke

If  $a$  and  $b$  are strings, then  $a + b$  requires string concatenation

```
In [3]: a, b = 'foo', 'bar'
In [4]: a + b
Out[4]: 'foobar'
```

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

```
In [5]: a, b = ['foo'], ['bar']
In [6]: a + b
Out[6]: ['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** Try executing the following in an Jupyter notebook cell

First,

```
import random
import numpy as np
```

Now try

```
%timeit
n = 100000
sum = 0
for i in range(n):
    x = random.uniform(0, 1)
    sum += x**2
```

(Note how `%%` in front of `timeit` converts this line magic into a cell magic for the notebook)

Followed by

```
%%timeit
n = 100000
x = np.random.uniform(0, 1, n)
np.sum(x**2)
```

You should find that the second code block — which achieves the same thing as the first — runs one to two orders of magnitude 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*

Although there are exceptions, 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 elementwise

For example, `np.cos` is a ufunc:

```
In [1]: import numpy as np
In [2]: np.cos(1.0)
Out[2]: 0.54030230586813977
In [3]: np.cos(np.linspace(0, 1, 3))
Out[3]: array([ 1.,  0.87758256,  0.54030231])
```

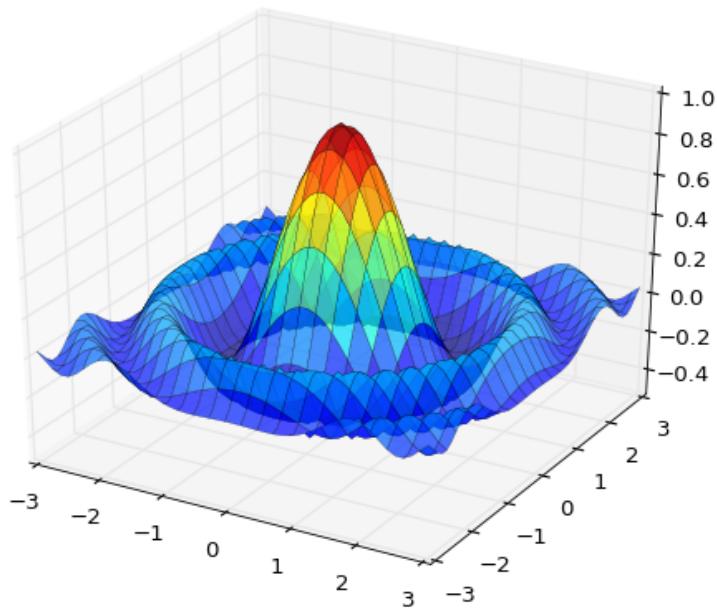
By exploiting ufuncs, many operations can be vectorized

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

```

import numpy as np
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

```

import numpy as np
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 add `%%timeit` to the top of these code snippets and run them in a notebook cell, you'll 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 very nice ways to speed up Python loops

## Numba

One of the most exciting developments in recent years in terms of scientific Python 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](#) and installed Anaconda, then you’ll be ready to use Numba

If not, try `import numba`

- If you get no complaints then you should be good to go
- If you do experience problems here or below then consider [installing Anaconda](#)

If you do have Anaconda installed, now might be a good time to run `conda update numba` from a system terminal

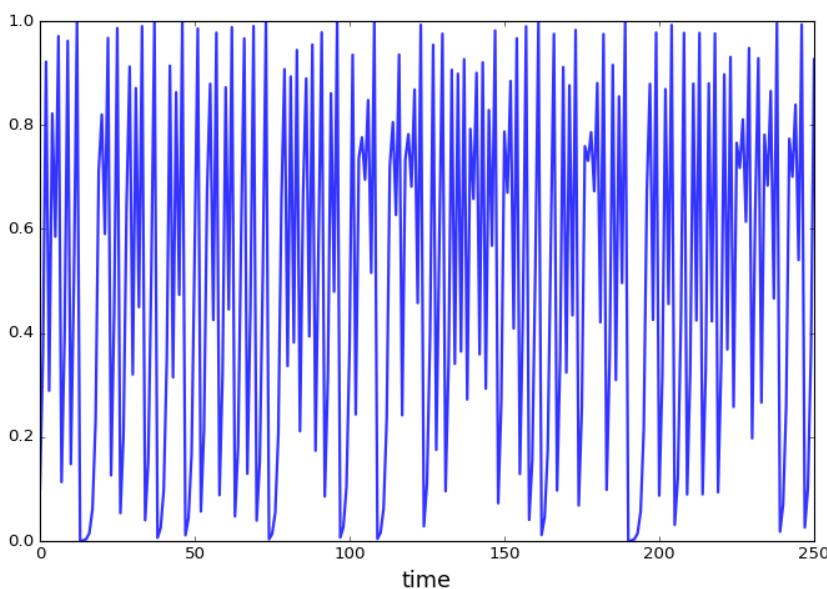
**An Example** Let’s consider some problems that are difficult to vectorize

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



Before starting let’s do some imports

```
from numba import jit
import numpy as np
```

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:

```
In [8]: timeit qm(0.1, int(10**5))
10 loops, best of 3: 65.7 ms per loop
```

```
In [9]: timeit qm_numba(0.1, int(10**5))
```

```
The slowest run took 434.34 times longer than the fastest. This could mean that an intermediate result is
1000 loops, best of 3: 260 µs per loop
```

Note the warning: The first execution is slower because of JIT compilation (see below)

Next time we get no such message

```
In [10]: timeit qm_numba(0.1, int(10**5))
1000 loops, best of 3: 259 µs per loop
```

```
In [11]: 65.7 * 1000 / 260 # Calculate speed gain
```

```
Out[11]: 252.69230769230768
```

We have produced a speed increase of two orders of magnitude

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

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

**Decorator Notation** If you don't need a separate name for the "numbaified" version of `qm`, you can just put `@jit` before the function

```
@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 extenstions — the wrapper code that interfaces between the resulting compiled code and Python

### Important Note:

In what follows we typically execute code in a Jupyter notebook

This is to take advantage of a Cython cell magic that makes Cython particularly easy to use

Some modifications are required to run the code outside a notebook

- See the book Cython by Kurt Smith or [the online documentation](#)

**A First Example** Let's start with a rather artificial example

Suppose that we want to compute the sum  $\sum_{i=0}^n \alpha^i$  for given  $\alpha, n$

Suppose further that we've forgotten the basic formula

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

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

**Python vs C** Here's a pure Python function that does the job

```
def geo_prog(alpha, n):
    current = 1.0
    sum = current
    for i in range(n):
        current = current * alpha
        sum = sum + current
    return sum
```

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

Here'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 an 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

```
In [21]: timeit geo_prog(0.99, int(10**6))
10 loops, best of 3: 101 ms per loop

In [22]: timeit geo_prog_cython(0.99, int(10**6))
10 loops, best of 3: 34.4 ms per loop
```

**Example 2: Cython with NumPy Arrays** Let's go back to the first problem that we worked with: generating the iterates of the quadratic map

$$x_{t+1} = 4x_t(1 - x_t)$$

The problem of computing iterates and returning a time series requires us to work with arrays

The natural array type to work with is NumPy arrays

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

```
%%cython
import numpy as np

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

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

```
In [24]: timeit qm_cython_first_pass(0.1, int(10**5))
10 loops, best of 3: 32.9 ms per loop
```

```
In [26]: timeit qm_numba(0.1, int(10**5))
1000 loops, best of 3: 259 µs per loop
```

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:

```
In [27]: timeit qm_cython(0.1, int(10**5))
1000 loops, best of 3: 452 µs per loop
```

This is pretty good, although not quite as fast as `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

### 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 extension modules from Fortran code using [F2Py](#)

F2Py is a Fortran-to-Python interface generator that is particularly simple to use

Robert Johansson provides a very nice [introduction](#) to F2Py, among other things

Recently, [an 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](#)

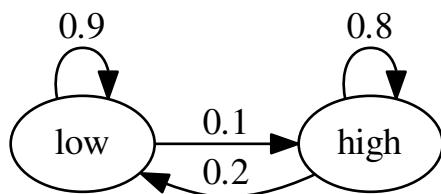
### Exercises

**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

[Solution notebook](#)

**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
- [Parakeet](#) : A runtime compiler aimed at scientific computing in Python
- [PyPy](#) : Runtime environment using just-in-time compiler
- [Nuitka](#) : Another Python compiler
- [Pyston](#) : Under development, sponsored by [Dropbox](#)

---

CHAPTER  
TWO

---

## INTRODUCTORY APPLICATIONS

This section of the course contains intermediate and foundational applications.

### Linear Algebra

#### Contents

- *Linear Algebra*
  - *Overview*
  - *Vectors*
  - *Matrices*
  - *Solving Systems of Equations*
  - *Eigenvalues and Eigenvectors*
  - *Further Topics*

#### 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,

$$\begin{aligned}y_1 &= a_{11}x_1 + a_{12}x_2 + \cdots + a_{1k}x_k \\&\vdots \\y_n &= a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nk}x_k\end{aligned}\tag{2.1}$$

The objective here is to solve for the “unknowns”  $x_1, \dots, x_k$  given  $a_{11}, \dots, a_{nk}$  and  $y_1, \dots, y_n$

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  $x = (x_1, \dots, x_n)$  or  $x = [x_1, \dots, x_n]$

We will write these sequences either horizontally or vertically as we please

(Later, when we wish to perform certain matrix operations, it will become necessary to distinguish between the two)

The set of all  $n$ -vectors is denoted by  $\mathbb{R}^n$

For example,  $\mathbb{R}^2$  is the plane, and a vector in  $\mathbb{R}^2$  is just a point in the plane

Traditionally, vectors are represented visually as arrows from the origin to the point

The following figure represents three vectors in this manner

If you’re interested, the Python code for producing this figure is [here](#)

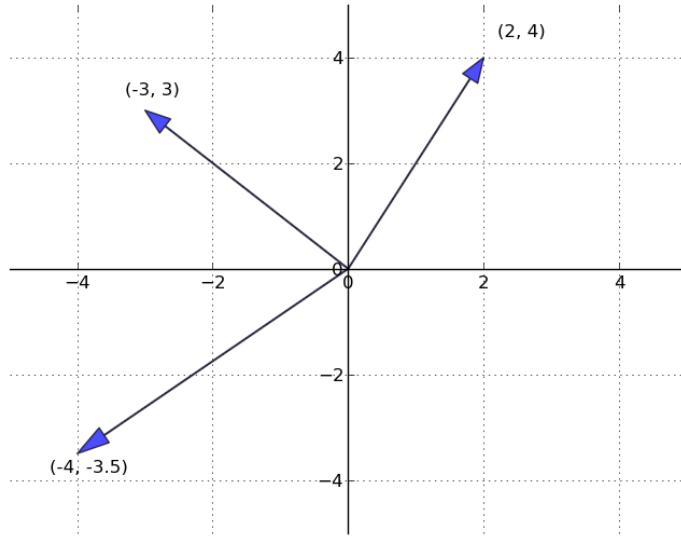
**Vector Operations** The two most common operators for vectors are addition and scalar multiplication, which we now describe

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  $\gamma$  and a vector  $x$  and produces

$$\gamma x := \begin{bmatrix} \gamma x_1 \\ \gamma x_2 \\ \vdots \\ \gamma x_n \end{bmatrix}$$



Scalar multiplication is illustrated in the next figure

In Python, a vector can be represented as a list or tuple, such as  $x = (2, 4, 6)$ , but is more commonly represented as a *NumPy array*

One advantage of NumPy arrays is that scalar multiplication and addition have very natural syntax

```
In [1]: import numpy as np

In [2]: x = np.ones(3)           # Vector of three ones

In [3]: y = np.array((2, 4, 6))  # Converts tuple (2, 4, 6) into array

In [4]: x + y
Out[4]: array([ 3.,  5.,  7.])

In [5]: 4 * x
Out[5]: array([ 4.,  4.,  4.])
```

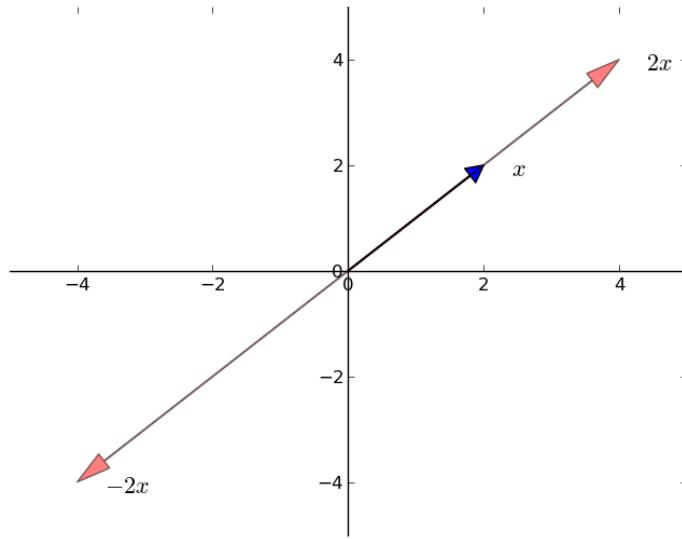
**Inner Product and Norm** The *inner product* of vectors  $x, y \in \mathbb{R}^n$  is defined as

$$x'y := \sum_{i=1}^n x_i y_i$$

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

```
In [6]: np.sum(x * y)           # Inner product of x and y
Out[6]: 12.0

In [7]: np.sqrt(np.sum(x**2))   # Norm of x, take one
Out[7]: 1.7320508075688772

In [8]: np.linalg.norm(x)       # Norm of x, take two
Out[8]: 1.7320508075688772
```

**Span** Given a set of vectors  $A := \{a_1, \dots, a_k\}$  in  $\mathbb{R}^n$ , it's natural to think about the new vectors we can create by performing linear operations

New vectors created in this manner are called *linear combinations* of  $A$

In particular,  $y \in \mathbb{R}^n$  is a linear combination of  $A := \{a_1, \dots, a_k\}$  if

$$y = \beta_1 a_1 + \cdots + \beta_k a_k \text{ for some scalars } \beta_1, \dots, \beta_k$$

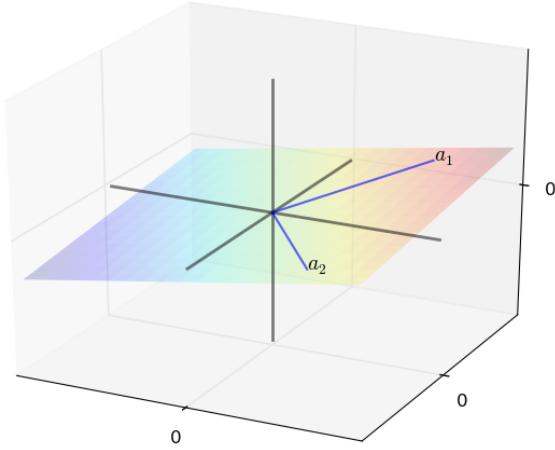
In this context, the values  $\beta_1, \dots, \beta_k$  are called the *coefficients* of the linear combination

The set of linear combinations of  $A$  is called the *span* of  $A$

The next figure shows the span of  $A = \{a_1, a_2\}$  in  $\mathbb{R}^3$

The span is a 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  $A := \{a_1, \dots, a_k\}$  in  $\mathbb{R}^n$  is said to be

- *linearly dependent* if some strict subset of  $A$  has the same span as  $A$
- *linearly independent* if it is not linearly dependent

Put differently, a set of vectors is linearly independent if no vector is redundant to the span, and linearly dependent otherwise

To illustrate the idea, recall *the figure* that showed the span of vectors  $\{a_1, a_2\}$  in  $\mathbb{R}^3$  as a plane through the origin

If we take a third vector  $a_3$  and form the set  $\{a_1, a_2, a_3\}$ , this set will be

- linearly dependent if  $a_3$  lies in the plane
- linearly independent otherwise

As another illustration of the concept, since  $\mathbb{R}^n$  can be spanned by  $n$  vectors (see the discussion of canonical basis vectors above), any collection of  $m > n$  vectors in  $\mathbb{R}^n$  must be linearly dependent

The following statements are equivalent to linear independence of  $A := \{a_1, \dots, a_k\} \subset \mathbb{R}^n$

1. No vector in  $A$  can be formed as a linear combination of the other elements
2. If  $\beta_1 a_1 + \dots + \beta_k a_k = 0$  for scalars  $\beta_1, \dots, \beta_k$ , then  $\beta_1 = \dots = \beta_k = 0$

(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  $A := \{a_1, \dots, a_k\} \subset \mathbb{R}^n$  is linearly independent and

$$y = \beta_1 a_1 + \dots + \beta_k a_k$$

then no other coefficient sequence  $\gamma_1, \dots, \gamma_k$  will produce the same vector  $y$

Indeed, if we also have  $y = \gamma_1 a_1 + \dots + \gamma_k a_k$ , then

$$(\beta_1 - \gamma_1) a_1 + \dots + (\beta_k - \gamma_k) a_k = 0$$

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  $i = 1, \dots, n$  are called the *principal diagonal*.  $A$  is called *diagonal* if the only nonzero entries are on the principal diagonal

If, in addition to being diagonal, each element along the principal diagonal is equal to 1, then  $A$  is called the *identity matrix*, and denoted by  $I$

**Matrix Operations** Just as was the case for vectors, a number of algebraic operations are defined for matrices

Scalar multiplication and addition are immediate generalizations of the vector case:

$$\gamma A = \gamma \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} := \begin{bmatrix} \gamma a_{11} & \cdots & \gamma a_{1k} \\ \vdots & \ddots & \vdots \\ \gamma a_{n1} & \cdots & \gamma a_{nk} \end{bmatrix}$$

and

$$A + B = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} + \begin{bmatrix} b_{11} & \cdots & b_{1k} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nk} \end{bmatrix} := \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1k} + b_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} + b_{n1} & \cdots & a_{nk} + b_{nk} \end{bmatrix}$$

In the latter case, the matrices must have the same shape in order for the definition to make sense

We also have a convention for *multiplying* two matrices

The rule for matrix multiplication generalizes the idea of inner products discussed above, and is designed to make multiplication play well with basic linear operations

If  $A$  and  $B$  are two matrices, then their product  $AB$  is formed by taking as its  $i, j$ -th element the inner product of the  $i$ -th row of  $A$  and the  $j$ -th column of  $B$

There are many tutorials to help you visualize this operation, such as [this one](#), or the discussion on the [Wikipedia page](#)

If  $A$  is  $n \times k$  and  $B$  is  $j \times m$ , then to multiply  $A$  and  $B$  we require  $k = j$ , and the resulting matrix  $AB$  is  $n \times m$

As perhaps the most important special case, consider multiplying  $n \times k$  matrix  $A$  and  $k \times 1$  column vector  $x$

According to the preceding rule, this gives us an  $n \times 1$  column vector

$$Ax = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} := \begin{bmatrix} a_{11}x_1 + \cdots + a_{1k}x_k \\ \vdots \\ a_{n1}x_1 + \cdots + a_{nk}x_k \end{bmatrix} \quad (2.2)$$

---

**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 <sup>1</sup>

You can create them manually from tuples of tuples (or lists of lists) as follows

```
In [1]: import numpy as np

In [2]: A = ((1, 2),
           ...:         (3, 4))

In [3]: type(A)
Out[3]: tuple

In [4]: A = np.array(A)

In [5]: type(A)
Out[5]: numpy.ndarray

In [6]: A.shape
Out[6]: (2, 2)
```

The `shape` attribute is a tuple giving the number of rows and columns — see *here* for more discussion

To get the transpose of  $A$ , use `A.transpose()` or, more simply, `A.T`

There are many convenient functions for creating common matrices (matrices of zeros, ones, etc.) — see *here*

Since operations are performed elementwise by default, scalar multiplication and addition have very natural syntax

```
In [8]: A = np.identity(3)

In [9]: B = np.ones((3, 3))

In [10]: 2 * A
Out[10]:
array([[ 2.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0.,  2.]])
```

---

<sup>1</sup> Although there is a specialized matrix data type defined in NumPy, it's more standard to work with ordinary NumPy arrays. See *this discussion*.

```
In [11]: A + B
Out[11]:
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  $f: \mathbb{R}^k \rightarrow \mathbb{R}^n$  is called *linear* if, for all  $x, y \in \mathbb{R}^k$  and all scalars  $\alpha, \beta$ , we have

$$f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$$

You can check that this holds for the function  $f(x) = Ax + b$  when  $b$  is the zero vector, and fails when  $b$  is nonzero

In fact, it's known that  $f$  is linear if and *only if* there exists a matrix  $A$  such that  $f(x) = Ax$  for all  $x$ .

### Solving Systems of Equations

Recall again the system of equations (2.1)

If we compare (2.1) and (2.2), we see that (2.1) can now be written more conveniently as

$$y = Ax \tag{2.3}$$

The problem we face is to determine a vector  $x \in \mathbb{R}^k$  that solves (2.3), 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  $y = f(x)$ ?

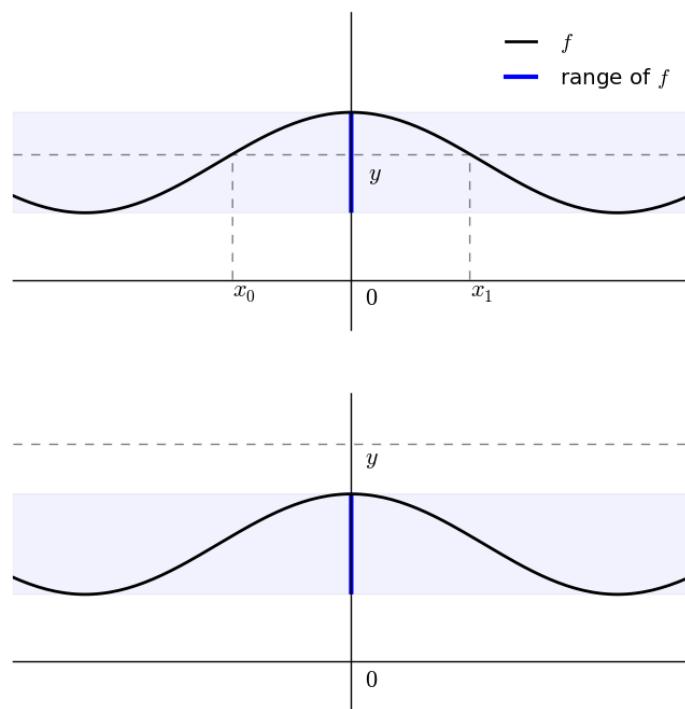
If so, is it always unique?

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 (2.3) that rule out these problems?

In this context, the most important thing to recognize about the expression  $Ax$  is that it corresponds to a linear combination of the columns of  $A$



In particular, if  $a_1, \dots, a_k$  are the columns of  $A$ , then

$$Ax = x_1a_1 + \cdots + x_ka_k$$

Hence the range of  $f(x) = Ax$  is exactly the span of the columns of  $A$

We want the range to be large, so that it contains arbitrary  $y$

As you might recall, the condition that we want for the span to be large is *linear independence*

A happy fact is that linear independence of the columns of  $A$  also gives us uniqueness

Indeed, it follows from our *earlier discussion* that if  $\{a_1, \dots, a_k\}$  are linearly independent and  $y = Ax = x_1a_1 + \cdots + x_ka_k$ , then no  $z \neq x$  satisfies  $y = Az$

**The  $n \times n$  Case** Let's discuss some more details, starting with the case where  $A$  is  $n \times n$

This is the familiar case where the number of unknowns equals the number of equations

For arbitrary  $y \in \mathbb{R}^n$ , we hope to find a unique  $x \in \mathbb{R}^n$  such that  $y = Ax$

In view of the observations immediately above, if the columns of  $A$  are linearly independent, then their span, and hence the range of  $f(x) = Ax$ , is all of  $\mathbb{R}^n$

Hence there always exists an  $x$  such that  $y = Ax$

Moreover, the solution is unique

In particular, the following are equivalent

1. The columns of  $A$  are linearly independent
2. For any  $y \in \mathbb{R}^n$ , the equation  $y = Ax$  has a unique solution

The property of having linearly independent columns is sometimes expressed as having *full column rank*

**Inverse Matrices** Can we give some sort of expression for the solution?

If  $y$  and  $A$  are scalar with  $A \neq 0$ , then the solution is  $x = A^{-1}y$

A similar expression is available in the matrix case

In particular, if square matrix  $A$  has full column rank, then it possesses a multiplicative *inverse matrix*  $A^{-1}$ , with the property that  $AA^{-1} = A^{-1}A = I$

As a consequence, if we pre-multiply both sides of  $y = Ax$  by  $A^{-1}$ , we get  $x = A^{-1}y$

This is the solution that 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  $y = Ax = x_1a_1 + x_2a_2 + x_3a_3$ , we can also write

$$y = x_1(\alpha a_2 + \beta a_3) + x_2a_2 + x_3a_3 = (x_1\alpha + x_2)a_2 + (x_1\beta + x_3)a_3$$

In other words, uniqueness fails

**Linear Equations with SciPy** Here's an illustration of how to solve linear equations with SciPy's `linalg` submodule

All of these routines are Python front ends to time-tested and highly optimized FORTRAN code

```
In [9]: import numpy as np

In [10]: from scipy.linalg import inv, solve, det

In [11]: A = ((1, 2), (3, 4))

In [12]: A = np.array(A)

In [13]: y = np.ones((2, 1)) # Column vector

In [14]: det(A) # Check that A is nonsingular, and hence invertible
Out[14]: -2.0

In [15]: A_inv = inv(A) # Compute the inverse

In [16]: A_inv
Out[16]:
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

In [17]: x = np.dot(A_inv, y) # Solution

In [18]: np.dot(A, x) # Should equal y
Out[18]:
array([[ 1.],
       [ 1.]])

In [19]: solve(A, y) # Produces same solution
Out[19]:
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  $\lambda$  is scalar and  $v$  is a non-zero vector in  $\mathbb{R}^n$  such that

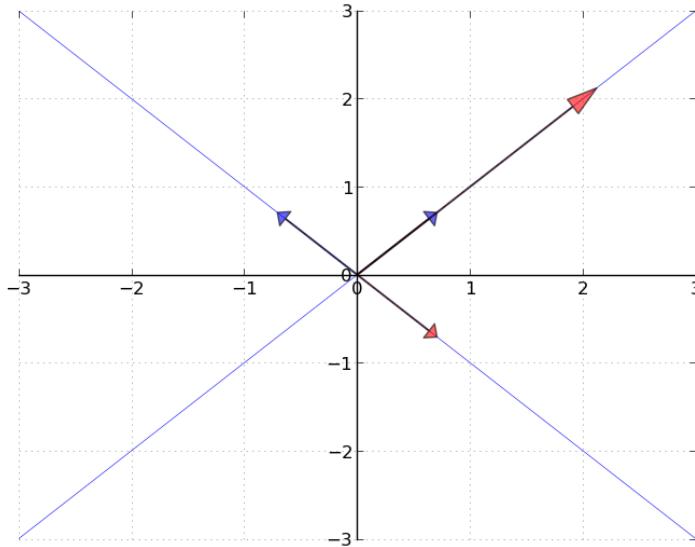
$$Av = \lambda v$$

then we say that  $\lambda$  is an *eigenvalue* of  $A$ , and  $v$  is an *eigenvector*

Thus, an eigenvector of  $A$  is a vector such that when the map  $f(x) = Ax$  is applied,  $v$  is merely scaled

The next figure shows two eigenvectors (blue arrows) and their images under  $A$  (red arrows)

As expected, the image  $Av$  of each  $v$  is just a scaled version of the original



The eigenvalue equation is equivalent to  $(A - \lambda I)v = 0$ , and this has a nonzero solution  $v$  only when the columns of  $A - \lambda I$  are linearly dependent

This in turn is equivalent to stating that the determinant is zero

Hence to find all eigenvalues, we can look for  $\lambda$  such that the determinant of  $A - \lambda I$  is zero

This problem can be expressed as one of solving for the roots of a polynomial in  $\lambda$  of degree  $n$

This in turn implies the existence of  $n$  solutions in the complex plane, although some might be repeated

Some nice facts about the eigenvalues of a square matrix  $A$  are as follows

1. The determinant of  $A$  equals the product of the eigenvalues
2. The trace of  $A$  (the sum of the elements on the principal diagonal) equals the sum of the eigenvalues
3. If  $A$  is symmetric, then all of its eigenvalues are real

4. If  $A$  is invertible and  $\lambda_1, \dots, \lambda_n$  are its eigenvalues, then the eigenvalues of  $A^{-1}$  are  $1/\lambda_1, \dots, 1/\lambda_n$

A corollary of the first statement is that a matrix is invertible if and only if all its eigenvalues are nonzero

Using SciPy, we can solve for the eigenvalues and eigenvectors of a matrix as follows

```
In [1]: import numpy as np
In [2]: from scipy.linalg import eig
In [3]: A = ((1, 2),
...:           (2, 1))
In [4]: A = np.array(A)
In [5]: evals, evecs = eig(A)
In [6]: evals
Out[6]: array([ 3.+0.j, -1.+0.j])
In [7]: evecs
Out[7]:
array([[ 0.70710678, -0.70710678],
       [ 0.70710678,  0.70710678]])
```

Note that the *columns* of `evecs` are the eigenvectors

Since any scalar multiple of an eigenvector is an eigenvector with the same eigenvalue (check it), the `eig` routine normalizes the length of each eigenvector to one

**Generalized Eigenvalues** It is sometimes useful to consider the *generalized eigenvalue problem*, which, for given matrices  $A$  and  $B$ , seeks generalized eigenvalues  $\lambda$  and eigenvectors  $v$  such that

$$Av = \lambda Bv$$

This can be solved in SciPy via `scipy.linalg.eig(A, B)`

Of course, if  $B$  is square and invertible, then we can treat the generalized eigenvalue problem as an ordinary eigenvalue problem  $B^{-1}Av = \lambda v$ , but this is not always the case

## 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<sup>2</sup>

**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

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k \quad (2.4)$$

**Spectral Radius** A result known as Gelfand's formula tells us that, for any square matrix  $A$ ,

$$\rho(A) = \lim_{k \rightarrow \infty} \|A^k\|^{1/k}$$

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 (2.4) is valid

**Positive Definite Matrices** Let  $A$  be a symmetric  $n \times n$  matrix

We say that  $A$  is

1. *positive definite* if  $x'Ax > 0$  for every  $x \in \mathbb{R}^n \setminus \{0\}$
2. *positive semi-definite* or *nonnegative definite* if  $x'Ax \geq 0$  for every  $x \in \mathbb{R}^n$

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

---

<sup>2</sup> 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.

- $B$  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.  $\frac{\partial y'Bz}{\partial B} = yz'$

**An Example** 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

*Question:* 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  $\lambda$  is an  $n \times 1$  vector of Lagrange multipliers

Try applying the above formulas 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  $\lambda$

Show that these conditions imply that

1.  $\lambda = -2Py$
2. The optimizing choice of  $u$  satisfies  $u = -(Q + B'PB)^{-1}B'PAx$
3. The function  $v(x) = -x'\tilde{P}x$  where  $\tilde{P} = A'PA - A'PB(Q + B'PB)^{-1}B'PA$

As we will see, in economic contexts Lagrange multipliers often are shadow prices

**Note:** If we don't care about the Lagrange multipliers, we can substitute the constraint into the objective function, and then just maximize  $-(Ax + Bu)'P(Ax + Bu) - u'Qu$  with respect to  $u$ . You can verify that this leads to the same maximizer.

**Further Reading** The documentation of the `scipy.linalg` submodule can be found [here](#)

Chapters 2 and 3 of the [following](#) text 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 read on linear algebra is [\[Janich94\]](#)

## 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 <sup>1</sup> 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  $\{x_1, \dots, x_n\}$

The set  $S$  is called the **state space** and  $x_1, \dots, x_n$  are the **state values**

A **Markov chain**  $\{X_t\}$  on  $S$  is a sequence of random variables on  $S$  that have the **Markov property**

This means that, for any date  $t$  and any state  $y \in S$ ,

$$\mathbb{P}\{X_{t+1} = y | X_t\} = \mathbb{P}\{X_{t+1} = y | X_t, X_{t-1}, \dots\} \quad (2.5)$$

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

$$P(x, y) := \mathbb{P}\{X_{t+1} = y | X_t = x\} \quad (x, y \in S) \quad (2.6)$$

By construction,

- $P(x, y)$  is the probability of going from  $x$  to  $y$  in one unit of time (one step)
- $P(x, \cdot)$  is the conditional distribution of  $X_{t+1}$  given  $X_t = x$

We can view  $P$  as a stochastic matrix where

$$P_{ij} = P(x_i, x_j) \quad 1 \leq i, j \leq n$$

Going the other way, if we take a stochastic matrix  $P$ , we can generate a Markov chain  $\{X_t\}$  as follows:

- draw  $X_0$  from some specified distribution
- for each  $t = 0, 1, \dots$ , draw  $X_{t+1}$  from  $P(X_t, \cdot)$

By construction, the resulting process satisfies (2.6)

---

<sup>1</sup> 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  $\alpha$  and  $\beta$ , we can address a range of questions, such as

- What is the average duration of unemployment?
- Over the long-run, what fraction of time does a worker find herself unemployed?
- Conditional on employment, what is the probability of becoming unemployed at least once over the next 12 months?

We'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

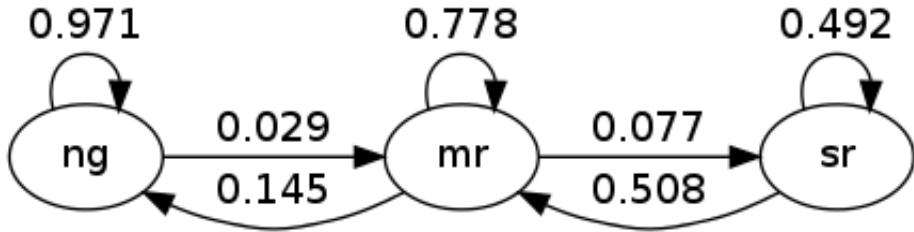
- the frequency is monthly
- the first state represents “normal growth”
- the second state represents “mild recession”
- the third state represents “severe recession”

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 this exercises we'll take the state space to be  $S = 0, \dots, n - 1$

**Rolling our own** To simulate a Markov chain, we need its stochastic matrix  $P$  and either an initial state or a probability distribution  $\psi$  for initial state to be drawn from

The Markov chain is then constructed as discussed above. To repeat:

1. At time  $t = 0$ , the  $X_0$  is set to some fixed state or chosen from  $\psi$
2. At each subsequent time  $t$ , the new state  $X_{t+1}$  is drawn from  $P(X_t, \cdot)$

In order to implement this simulation procedure, we need a method for generating draws from a discrete distributions

For this task we'll use [DiscreteRV](#) from [QuantEcon](#)

```

In [64]: from quantecon import DiscreteRV
In [65]: psi = (0.1, 0.9)                      # Probabilities over sample space {0, 1}
In [66]: d = DiscreteRV(psi)
In [67]: d.draw(5)                                # Generate 5 independent draws from psi
Out[67]: 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  $\text{init}$

- A positive integer `sample_size` representing the length of the time series the function should return

```
import numpy as np
import quantecon as qe

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

$$P := \begin{pmatrix} 0.4 & 0.6 \\ 0.2 & 0.8 \end{pmatrix} \quad (2.7)$$

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

```
In [5]: P = [[0.4, 0.6], [0.2, 0.8]]
In [6]: X = mc_sample_path(P, sample_size=100000)
In [7]: np.mean(X == 0)
Out[7]: 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

```
In [6]: import numpy as np
In [7]: import quantecon as qe
In [8]: P = [[0.4, 0.6], [0.2, 0.8]]
In [9]: mc = qe.MarkovChain(P)
```

```
In [10]: X = mc.simulate(ts_length=1000000)
```

```
In [11]: np.mean(X == 0)
Out[11]: 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)

```
In [17]: %timeit mc_sample_path(P, sample_size=1000000) # our version
1 loops, best of 3: 6.86 s per loop
```

```
In [18]: %timeit mc.simulate(ts_length=1000000) # qe version
10 loops, best of 3: 72.5 ms per loop
```

**Adding state values** If we wish to, we can provide a specification of state values to `MarkovChain`

These state values can be integers, floats, or even strings

The following code illustrates

```
In [9]: mc = qe.MarkovChain(P, state_values=('employed', 'unemployed'))
```

```
In [10]: mc.simulate(ts_length=4)
Out[10]: array(['employed', 'unemployed', 'employed', 'employed'],
              dtype='|<U10')
```

If we want indices rather than state values we can use

```
In [11]: mc.simulate_indices(ts_length=4)
Out[11]: 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  $\psi_t$

What then is the distribution of  $X_{t+1}$ , or, more generally, of  $X_{t+m}$ ?

**Solution** Let  $\psi_t$  be the distribution of  $X_t$  for  $t = 0, 1, 2, \dots$

Our first aim is to find  $\psi_{t+1}$  given  $\psi_t$  and  $P$

To begin, pick any  $y \in S$

Using the [law of total probability](#), we can decompose the probability that  $X_{t+1} = y$  as follows:

$$\mathbb{P}\{X_{t+1} = y\} = \sum_{x \in S} \mathbb{P}\{X_{t+1} = y \mid X_t = x\} \cdot \mathbb{P}\{X_t = x\}$$

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  $\psi_{t+1}$  and  $\psi_t$  as *row vectors* (as is traditional in this literature), these  $n$  equations are summarized by the matrix expression

$$\psi_{t+1} = \psi_t P \quad (2.8)$$

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 (2.8), the expression  $\psi_{t+m} = \psi_t P^m$  is also valid — here  $P^m$  is the  $m$ -th power of  $P$ . As a special case, we see that if  $\psi_0$  is the initial distribution from which  $X_0$  is drawn, then  $\psi_0 P^m$  is the distribution of  $X_m$ .

This is very important, so let's repeat it

$$X_0 \sim \psi_0 \implies X_m \sim \psi_0 P^m \quad (2.9)$$

and, more generally,

$$X_t \sim \psi_t \implies X_{t+m} \sim \psi_t P^m \quad (2.10)$$

**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 (2.10), but now with  $\psi_t$  putting all probability on state  $x$

- 1 in the  $x$ -th position and zero elsewhere

Inserting this into (2.10), 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  $\psi$  be the current *cross-sectional* distribution over  $\{0, 1\}$

- For example,  $\psi(0)$  is the unemployment rate

The cross-sectional distribution records the fractions of workers employed and unemployed at a given moment

The same distribution also describes the fractions of a particular worker's career spent being employed and unemployed, respectively

## Irreducibility and Aperiodicity

Irreducibility and aperiodicity are central concepts of modern Markov chain theory

Let's see what they're about

**Irreducibility** Let  $P$  be a fixed stochastic matrix

Two states  $x$  and  $y$  are said to **communicate** with each other if there exist positive integers  $j$  and  $k$  such that

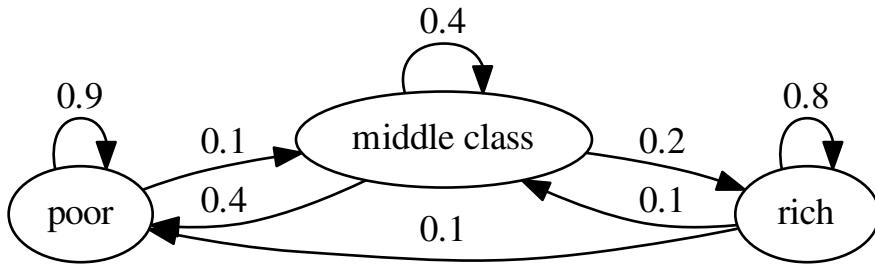
$$P^j(x, y) > 0 \quad \text{and} \quad P^k(y, x) > 0$$

In view of our discussion *above*, this means precisely that

- state  $x$  can be reached eventually from state  $y$ , and
- state  $y$  can be reached eventually from state  $x$

The stochastic matrix  $P$  is called **irreducible** if all states communicate; that is, if  $x$  and  $y$  communicate for all  $(x, y)$  in  $S \times S$

For example, consider the following transition probabilities for wealth of a fictitious set of households



We can translate this into a stochastic matrix, putting zeros where there's no edge between nodes

$$P := \begin{pmatrix} 0.9 & 0.1 & 0 \\ 0.4 & 0.4 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{pmatrix}$$

It's clear from the graph that this stochastic matrix is irreducible: we can reach any state from any other state eventually

We can also test this using QuantEcon's `MarkovChain` class

```

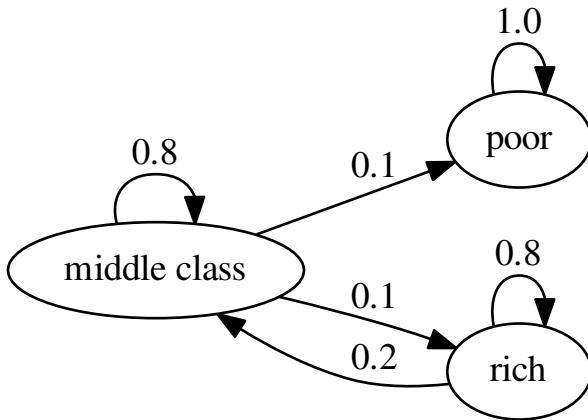
In [1]: import quantecon as qe

In [2]: P = [[0.9, 0.1, 0.0],
...           [0.4, 0.4, 0.2],
...           [0.1, 0.1, 0.8]]

In [3]: mc = qe.MarkovChain(P, ('poor', 'middle', 'rich'))

In [4]: mc.is_irreducible
Out[4]: True
  
```

Here's a more pessimistic scenario, where the poor are poor forever



This stochastic matrix is not irreducible, since, for example, *rich* is not accessible from *poor*

Let's confirm this

```

In [5]: P = [[1.0, 0.0, 0.0],
...:      [0.1, 0.8, 0.1],
...:      [0.0, 0.2, 0.8]]
In [6]: mc = qe.MarkovChain(P, ('poor', 'middle', 'rich'))
In [7]: mc.is_irreducible
Out[7]: False
  
```

We can also determine the “communication classes”

```

In [8]: mc.communication_classes
Out[8]:
[array(['poor'], dtype='|<U6'),
 array(['middle', 'rich'], dtype='|<U6')]
  
```

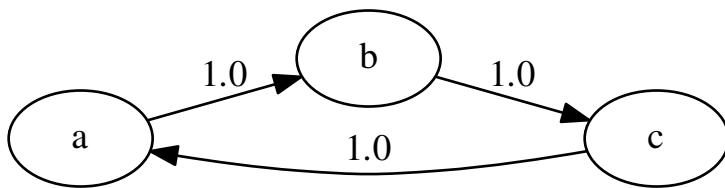
It might be clear to you already that irreducibility is going to be important in terms of long run outcomes

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 predictable way, and aperiodic otherwise

Here's a trivial example with three states



The chain cycles with period 3:

```

In [2]: P = [[0, 1, 0],
...:      [0, 0, 1],
...:      [1, 0, 0]]
In [3]: mc = qe.MarkovChain(P)
In [4]: mc.period
Out[4]: 3

```

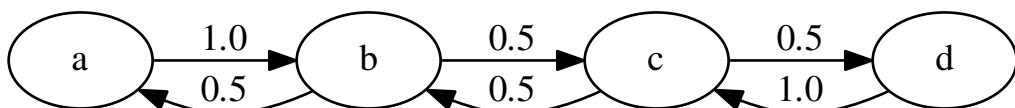
More formally, the **period** of a state  $x$  is the greatest common divisor of the set of integers

$$D(x) := \{j \geq 1 : P^j(x, x) > 0\}$$

In the last example,  $D(x) = \{3, 6, 9, \dots\}$  for every state  $x$ , so the period is 3

A stochastic matrix is called **aperiodic** if the period of every state is 1, and **periodic** otherwise

For example, the stochastic matrix associated with the transition probabilities below is periodic because, for example, state  $a$  has period 2



We can confirm that the stochastic matrix is periodic as follows

```

In [1]: import quantecon as qe
In [2]: 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]]

```

```
In [3]: mc = qe.MarkovChain(P)
```

```
In [4]: mc.period
```

```
Out[4]: 2
```

```
In [5]: mc.is_aperiodic
```

```
Out[5]: False
```

## Stationary Distributions

As seen in (2.8), we can shift probabilities forward one unit of time via postmultiplication by  $P$

Some distributions are invariant under this updating process — for example,

```
In [1]: import numpy as np
```

```
In [2]: P = np.array([[.4, .6], [.2, .8]])
```

```
In [3]: psi = (0.25, 0.75)
```

```
In [4]: np.dot(psi, P)
```

```
Out[4]: array([ 0.25,  0.75])
```

Such distributions are called **stationary**, or **invariant**. Formally, a distribution  $\psi^*$  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  $\psi^*$
2. For any initial distribution  $\psi_0$ , we have  $\|\psi_0 P^t - \psi^*\| \rightarrow 0$  as  $t \rightarrow \infty$

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  $\beta \rightarrow 0$ , and to one as  $\alpha \rightarrow 0$

**Calculating Stationary Distributions** As discussed above, a given Markov matrix  $P$  can have many stationary distributions

That is, there can be many row vectors  $\psi$  such that  $\psi = \psi P$

In fact if  $P$  has two distinct stationary distributions  $\psi_1, \psi_2$  then it has infinitely many, since in this case, as you can verify,

$$\psi_3 := \lambda \psi_1 + (1 - \lambda) \psi_2$$

is a stationary distribution for  $P$  for any  $\lambda \in [0, 1]$

If we restrict attention to the case where only one stationary distribution exists, one option for finding it is to try to solve the linear system  $\psi(I_n - P) = 0$  for  $\psi$ , where  $I_n$  is the  $n \times n$  identity

But the zero vector solves this equation

Hence we need to impose the restriction that the solution must be a probability distribution

A suitable algorithm is implemented in QuantEcon — the next code block illustrates

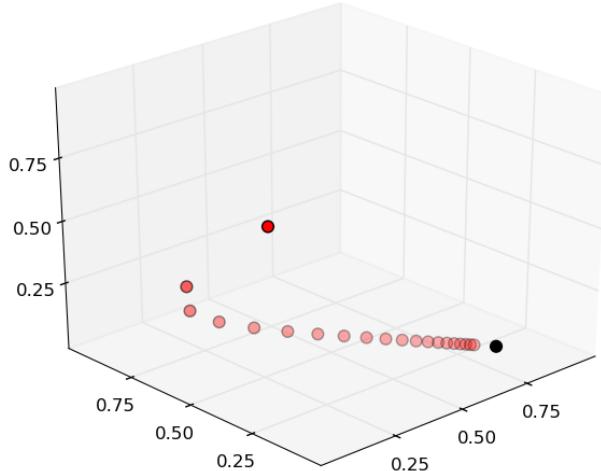
```
In [2]: import quantecon as qe
In [3]: P = [[0.4, 0.6], [0.2, 0.8]]
In [4]: mc = qe.MarkovChain(P)
In [5]: mc.stationary_distributions # Show all stationary distributions
Out[5]: array([[ 0.25,  0.75]])
```

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  $\psi^*$  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  $\psi$ , represented as a vector in  $\mathbb{R}^3$
- The other red dots are the distributions  $\psi P^t$  for  $t = 1, 2, \dots$
- The black dot is  $\psi^*$

The code for the figure can be found in the QuantEcon applications library — you might like to try experimenting with different initial conditions

### Ergodicity

Under irreducibility, yet another important result obtains: For all  $x \in S$ ,

$$\frac{1}{n} \sum_{t=1}^n \mathbf{1}\{X_t = x\} \rightarrow \psi^*(x) \quad \text{as } n \rightarrow \infty \quad (2.11)$$

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 (2.11) is valid.

The convergence in (2.11) 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

$$\mathbb{E}[h(X_t)] \quad (2.12)$$

and conditional expectations such as

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] \quad (2.13)$$

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 (2.12) 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  $\psi$  is the distribution of  $X_0$

Since  $\psi$  and hence  $\psi P^t$  are row vectors, we can also write this as

$$\mathbb{E}[h(X_t)] = \psi P^t h$$

For the conditional expectation (2.13), 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

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] = (P^k h)(x) \quad (2.14)$$

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

$$(I - \beta P)^{-1} = I + \beta P + \beta^2 P^2 + \dots$$

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  $\bar{X}_n \rightarrow p$  as  $n \rightarrow \infty$ , where

$$\bar{X}_n := \frac{1}{n} \sum_{t=1}^n \mathbf{1}\{X_t = 0\}$$

Your exercise is to illustrate this convergence

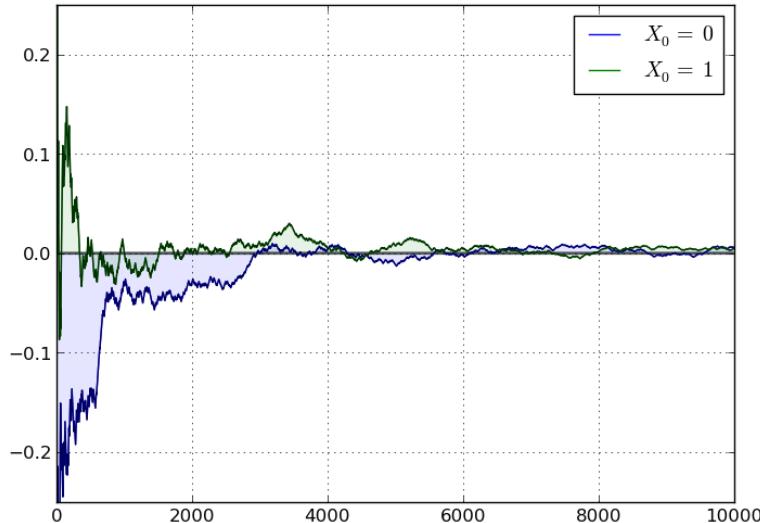
First,

- 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

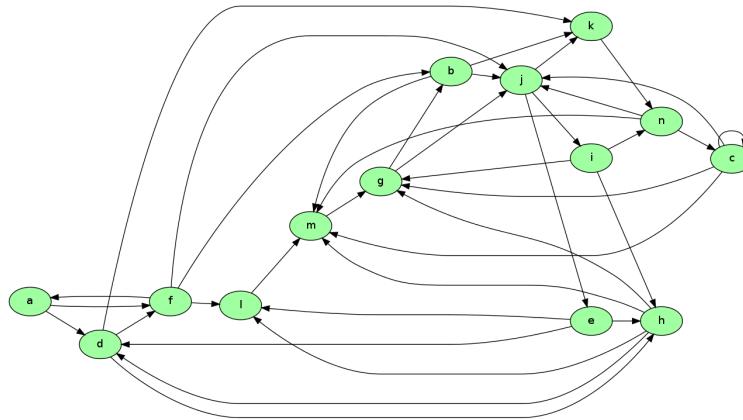
1. receive a large set of accurate matches

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

- $\ell_i$  is the total number of outbound links from  $i$
  - $L_j$  is the set of all pages  $i$  such that  $i$  has a link to  $j$

This is a measure of the number of inbound links, weighted by their own ranking (and normalized by  $1/\ell_i$ )

There is, however, another interpretation, and it brings us back to Markov chains

Let  $P$  be the matrix given by  $P(i, j) = \mathbf{1}\{i \rightarrow j\}/\ell_i$  where  $\mathbf{1}\{i \rightarrow j\} = 1$  if  $i$  has a link to  $j$  and zero otherwise

The matrix  $P$  is a stochastic matrix provided that each page has at least one link

With this definition of  $P$  we have

$$r_j = \sum_{i \in L_j} \frac{r_i}{\ell_i} = \sum_{\text{all } i} \mathbf{1}\{i \rightarrow j\} \frac{r_i}{\ell_i} = \sum_{\text{all } i} P(i, j) r_i$$

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 from the [main repository](#) — 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

```
In [1]: import re
In [2]: re.findall('\w+', 'x +++ y ***** z') # \w matches alphanumerics
Out[2]: ['x', 'y', 'z']
```

```
In [3]: re.findall('\w', 'a ^ b && $$ c')
Out[3]: ['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  $\{x_0, \dots, x_{n-1}\} \subset \mathbb{R}$  and a stochastic  $n \times n$  matrix  $P$  such that

- $x_0 = -m \sigma_y$
- $x_{n-1} = m \sigma_y$
- $x_{i+1} = x_i + s$  where  $s = (x_{n-1} - x_0)/(n - 1)$

Let  $F$  be the cumulative distribution function of the normal distribution  $N(0, \sigma_u^2)$

The values  $P(x_i, x_j)$  are computed to approximate the AR(1) process — omitting the derivation, the rules are as follows:

1. If  $j = 0$ , then set

$$P(x_i, x_0) = P(x_i, x_0) = F(x_0 - \rho x_i + s/2)$$

2. If  $j = n - 1$ , then set

$$P(x_i, x_{n-1}) = P(x_i, x_{n-1}) = 1 - F(x_{n-1} - \rho x_i - s/2)$$

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  $\{x_0, \dots, x_{n-1}\} \subset \mathbb{R}$  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

[Solution notebook](#)

# Orthogonal Projection and its Applications

## Contents

- *Orthogonal Projection and its Applications*
  - *Overview*
  - *Key Definitions*
  - *The Orthogonal Projection Theorem*
  - *Orthonormal Bases*
  - *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. This include, but are not limited to,

- Least squares and linear regression
- Conditional expectation
- Gram–Schmidt orthogonalization
- QR decomposition
- Orthogonal polynomials
- etc

In this lecture we focus on

- key results
- standard applications such as least squares regression

**Further Reading** For background and foundational concepts, see our lecture on linear algebra

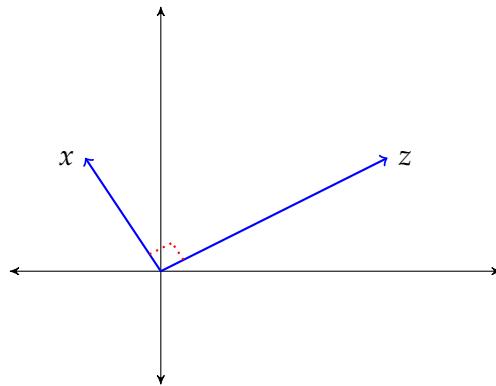
For more proofs and greater theoretical detail, see [A Primer in Econometric Theory](#)

For a complete set of proofs in a general setting, see, for example, [\[Rom05\]](#)

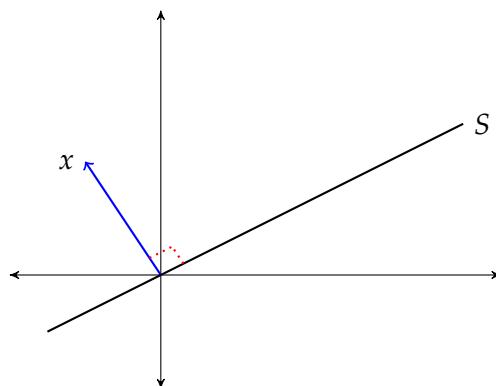
For an advanced treatment of projection in the context of least squares prediction, see [this book chapter](#)

### Key Definitions

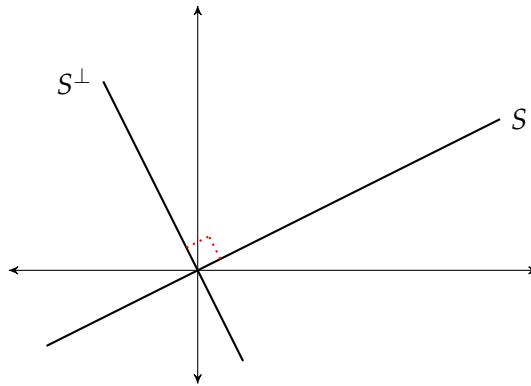
If  $x, z \in \mathbb{R}^n$  and  $\langle x, z \rangle = 0$  then  $x$  and  $z$  are said to be **orthogonal**, and we write  $x \perp z$



Given  $S \subset \mathbb{R}^n$ , 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  $S$  is the set  $S^\perp := \{x \in \mathbb{R}^n : x \perp S\}$



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

To see this, fix  $x, y \in S^\perp$  and  $\alpha, \beta \in \mathbb{R}$

Observe that if  $z \in S$ , then

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

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

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

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

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

In the case of  $k = 2$  this is easy to see, since orthogonality gives

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

**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

The problem considered by the orthogonal projection theorem (OPT) is to find the closest approximation to an arbitrary vector from within a given linear subspace

The theorem, stated below, tells us that this problem always has a unique solution, and provides a very useful characterization

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

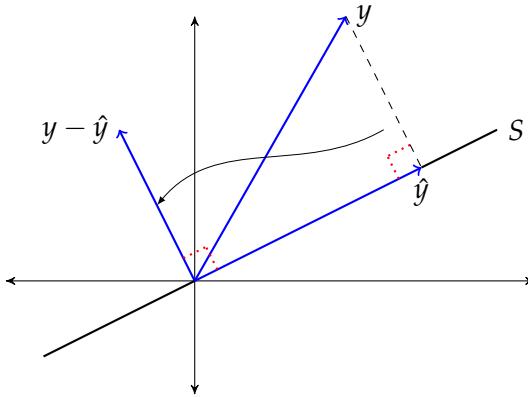
$$\hat{y} := \arg \min_{z \in S} \|y - z\|$$

Moreover, the solution  $\hat{y}$  is the unique vector in  $\mathbb{R}^n$  such that

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

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

The next figure provides some intuition



We'll omit the full proof but let's at least cover sufficiency of the 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  $y - \hat{y} \perp S$

Letting  $z$  be any other point in  $S$  and using the fact that  $S$  is a linear subspace, we have

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

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

**Orthogonal Projection as a Mapping** Holding  $S$  fixed, we have a functional relationship

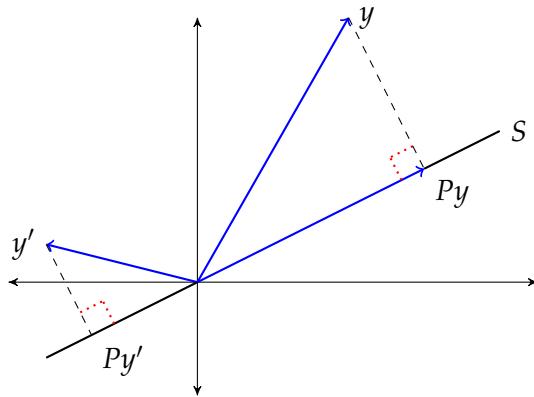
$$y \mapsto \text{its orthogonal projection } \hat{y} \in S$$

By the OPT, this is a well-defined mapping from  $\mathbb{R}^n$  to  $\mathbb{R}^n$

In what follows it is denoted by  $P$

- $Py$  represents the projection  $\hat{y}$
- We write  $P = \text{proj } S$

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 properties, such as

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

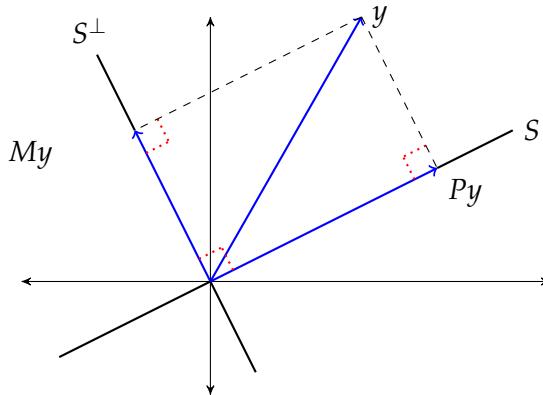
For example, to prove 1, observe that  $y = Py + y - Py$  and apply the Pythagorean law

**The Residual Projection** Here's another version of the OPT

**Theorem.** If  $S$  is a linear subspace of  $\mathbb{R}^n$ ,  $P = \text{proj } S$  and  $M = \text{proj } S^\perp$ , 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 Bases

An orthogonal set  $O \subset \mathbb{R}^n$  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  $\text{span } O = S$ , then  $O$  is called an **orthonormal basis** of  $S$

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

One example of an orthonormal set is the canonical basis  $\{e_1, \dots, e_n\}$ , which forms an orthonormal basis of  $\mathbb{R}^n$

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

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

To see this, observe that since  $x \in \text{span}\{u_1, \dots, u_k\}$ , we can find scalars  $\alpha_1, \dots, \alpha_k$  s.t.

$$x = \sum_{j=1}^k \alpha_j u_j \tag{2.15}$$

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 (2.15) verifies the claim

**Projection onto an Orthonormal Basis** When we have an orthonormal basis for the subspace we are projecting onto, computing the projection is straightforward:

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

$$Py = \sum_{i=1}^k \langle y, u_i \rangle u_i, \quad \forall y \in \mathbb{R}^n \tag{2.16}$$

Proof: Fix  $y \in \mathbb{R}^n$  and let  $Py$  be as defined in (2.16)

Clearly,  $Py \in S$

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

It suffices to show that  $y - Py \perp$  any basis element (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

It is not too difficult to show that if  $S$  is any linear subspace of  $\mathbb{R}^n$  and  $P = \text{proj } S$ , then  $P$  is a linear function from  $\mathbb{R}^n$  to  $\mathbb{R}^n$

It follows that  $P = \text{proj } S$  can be represented as a matrix

Below we use  $P$  for both the orthogonal projection mapping and the matrix that represents it

But what does the matrix look like?

**Theorem.** If  $P = \text{proj } S$  and 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.  $y - Py \perp S$

Here 1 is true because

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

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

On the other hand, 2 is equivalent to the statement

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

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

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

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

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

From the preceding theorem,  $P = X(X'X)^{-1}X'$  projects onto  $S$

In this context,  $P = \text{proj } S$  is often called the **projection matrix**

- The matrix  $M = I - P$  satisfies  $M = \text{proj}(S^\perp)$  and is sometimes called the **annihilator**

As a further illustration of the last result, suppose that  $U$  is  $n \times k$  with orthonormal columns

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

We know that the projection of  $y$  onto  $S$  is

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

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

Hence

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

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

**Application: Overdetermined Systems of Equations** Consider linear system  $Xb = y$  where  $y \in \mathbb{R}^n$  and  $X$  is  $n \times k$  with linearly independent columns

Given  $X$  and  $y$ , we seek  $b \in \mathbb{R}^k$  satisfying this equation

If  $n > k$  (more equations than unknowns), then the system 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 is based around the OPT

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

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

Proof: Note that

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

Since  $Py$  is the orthogonal projection onto  $\text{span}(X)$  we have

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

In other words,

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

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 insight on many fundamental geometric and theoretical properties of linear regression

We treat only some of the main ideas

**The Setting** Here's one way to introduce linear regression

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

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

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

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

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

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

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

The theory of statistical learning tells us we should take it to be relatively simple to prevent overfitting

If we let  $\mathcal{F}$  be the class of linear functions and drop the constant  $1/N$ , the problem is

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

This is the **linear least squares problem**

**Solution** To switch to matrix notation, define

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

and

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 increasing transforms don't affect minimizers we have

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

By our results above on overdetermined systems, 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  $:= \|\hat{y}\|^2$

It's well known that  $TSS = ESS + SSR$  always holds

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

Many other standards results about least squares regression follow easily from the OPT

### Orthogonalization and Decomposition

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

The main result of interest is a famous algorithm for generating orthonormal sets from linearly independent sets

The next section gives details

**Gram-Schmidt Orthogonalization Theorem** For each linearly independent set  $\{x_1, \dots, x_k\} \subset \mathbb{R}^n$ , there exists an orthonormal set  $\{u_1, \dots, u_k\}$  with

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

Construction uses the **Gram-Schmidt orthogonalization** procedure

One version of this procedure is as follows: For  $i = 1, \dots, k$ , set

- $S_i := \text{span}\{x_1, \dots, x_i\}$  and  $M_i := \text{proj } S_i^\perp$
- $v_i := M_{i-1}x_i$  where  $M_0$  is the identity mapping
- $u_i := v_i / \|v_i\|$

The sequence  $u_1, \dots, u_k$  has the stated properties

In the exercises below you are asked to implement this algorithm and test it using projection

**QR Decomposition** Here's a well known result that uses the preceding algorithm to produce a useful decomposition

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

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

Proof sketch: Let

- $x_j := \text{col}_j(X)$

- $\{u_1, \dots, u_k\}$  be orthonormal with same span as  $\{x_1, \dots, x_k\}$  (to be constructed using Gram-Schmidt)
- $Q$  be formed from cols  $u_i$

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

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

Some rearranging gives  $X = QR$

**Linear Regression via QR Decomposition** For  $X$  and  $y$  as above we have  $\hat{\beta} = (X'X)^{-1}X'y$

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  $S \subset \mathbb{R}^n$ , we have  $S \cap S^\perp = \{0\}$

**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

[Solution notebook](#)

## 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

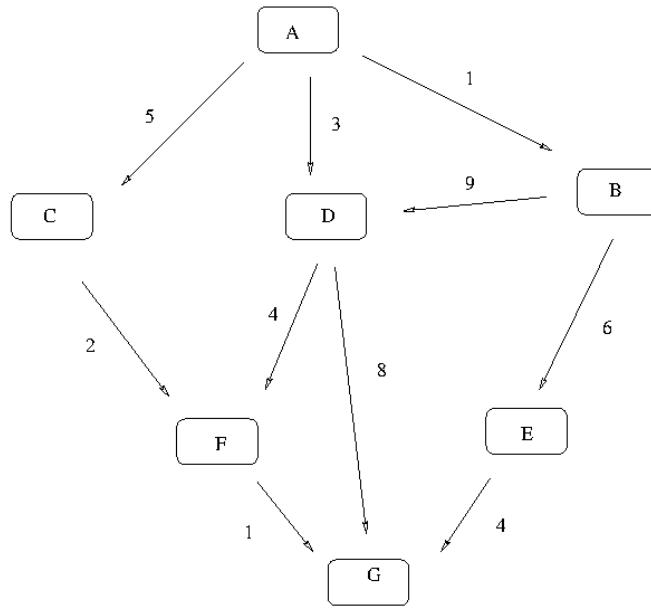
- Arrows (edges) indicate the movements we can take
- Numbers next to edges indicate the cost of traveling that edge

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

$$\min_{w \in F_v} \{c(v, w) + J(w)\} \quad (2.17)$$

where

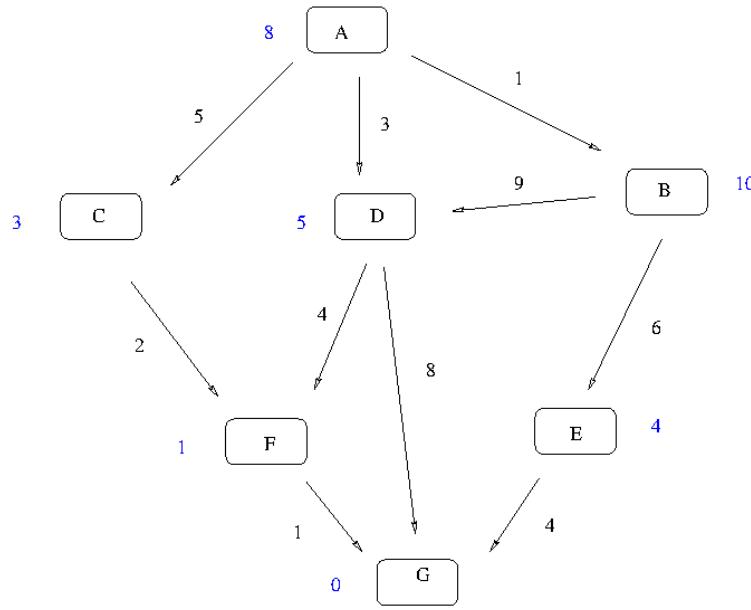
- $F_v$  is the set of nodes that can be reached from  $v$  in one step
- $c(v, w)$  is the cost of traveling from  $v$  to  $w$

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

$$J(v) = \min_{w \in F_v} \{c(v, w) + J(w)\} \quad (2.18)$$



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

$$J_0(v) = M \text{ if } v \neq \text{ destination, else } J_0(v) = 0 \quad (2.19)$$

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

[Solution notebook](#)

# 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 symbol) 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(y_t) \quad (2.20)$$

which represents the expected value of the discounted utility of his income

The constant  $\beta$  lies in  $(0, 1)$  and is called a **discount factor**

The smaller is  $\beta$ , the more the worker discounts future utility relative to current utility

The variable  $y_t$  is

- 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  $\alpha$  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

- assuming that you know the answer, then
- writing down some natural conditions that the answer must satisfy, then
- 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 (2.20) 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

$$V(w) = u(w) + \beta[(1 - \alpha)V(w) + \alpha U] \quad (2.21)$$

and

$$U = u(c) + \beta \sum_{w'} \max \{U, V(w')\} p(w') \quad (2.22)$$

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 (2.23) and (2.24) are the values of a worker in a particular situation *today*
- The right hand sides of the equations are the discounted (by  $\beta$ ) expected values of the possible situations that worker can be in *tomorrow*
- But *tomorrow* the worker can be in only one of the situations whose values *today* are on the left sides of our two equations

Equation (2.24) incorporates the fact that a currently unemployed worker knows that if his next period wage offer is  $w'$ , he will choose to remain unemployed unless  $U < V(w')$

Equations (2.23) and (2.24) are called *Bellman equations* after the mathematician Richard Bellman

It turns out that equations (2.23) and (2.24) 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  $\gamma$

If our worker does receive an offer, the wage offer is drawn from  $p$  as before

He either accepts or rejects the offer

Otherwise the model is the same

With some thought, you will be able to convince yourself that  $V$  and  $U$  should now satisfy

$$V(w) = u(w) + \beta[(1 - \alpha)V(w) + \alpha U] \quad (2.23)$$

and

$$U = u(c) + \beta(1 - \gamma)U + \beta\gamma \sum_{w'} \max\{U, V(w')\} p(w') \quad (2.24)$$

**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 (2.23) and (2.24)
3. update the left hand sides from this rule and then repeat

In other words, we are iterating using the rules

$$V_{n+1}(w) = u(w) + \beta[(1 - \alpha)V_n(w) + \alpha U_n] \quad (2.25)$$

and

$$U_{n+1} = u(c) + \beta(1 - \gamma)U_n + \beta\gamma \sum_{w'} \max\{U_n, V_n(w')\} p(w') \quad (2.26)$$

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 (2.23) and (2.24)

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](#) from the [applications repository](#)

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 approach is to iterate until successive iterates are closer together than some small tolerance level

We then return the current iterate as an approximate solution

Let's plot the approximate solutions  $U$  and  $V$  to see what they look like

We'll use the default parameterizations found in the code above

```
"""
Generate plots of value of employment and unemployment in the McCall model.

"""

import numpy as np
import matplotlib.pyplot as plt

from mccall_bellman_iteration import McCallModel, solve_mccall_model

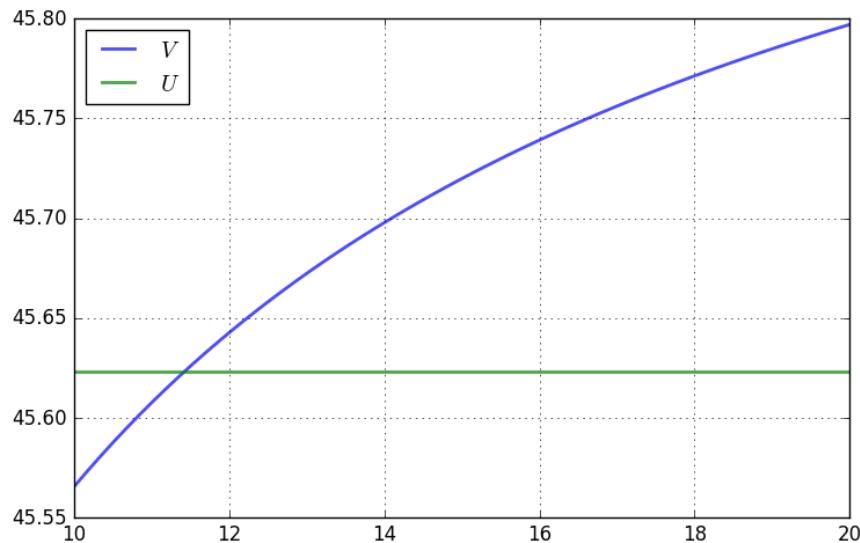
mcm = McCallModel()
V, U = solve_mccall_model(mcm)

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.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

We've written a function called `compute_reservation_wage` that takes an instance of a McCall model and returns the reservation wage associated with a given model

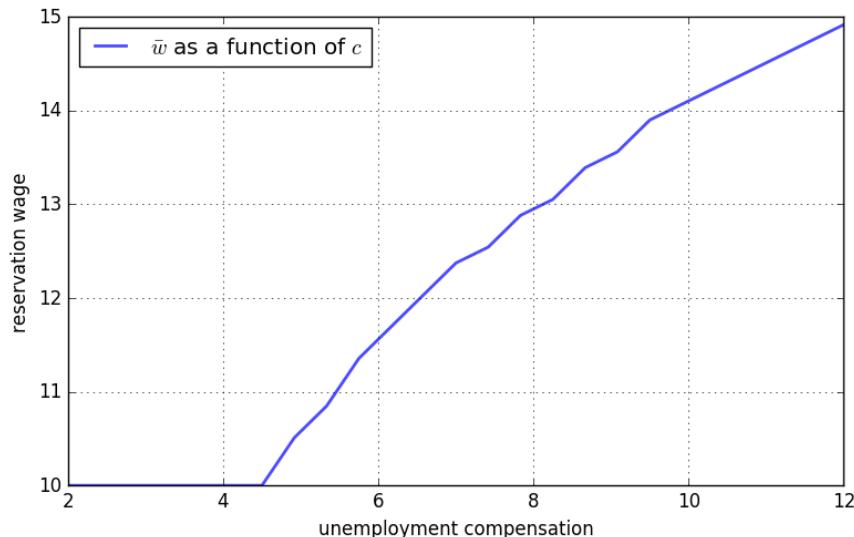
If  $V(w) < U$  for all  $w$ , then the function returns `np.inf`

Below you'll be asked to try to produce a version of this function as an exercise

For now let's use it to look at how the reservation wage varies with parameters

**The Reservation Wage and Unemployment Compensation** First, let's look at how  $\bar{w}$  varies with unemployment compensation

In the figure below, we use the default parameters in the `McCallModel` class, apart from  $c$  (which takes the values given on the horizontal axis)



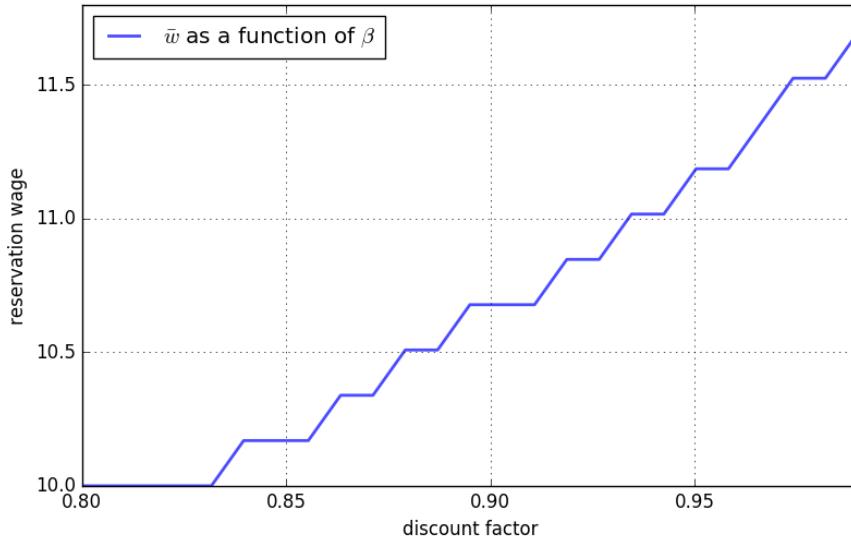
As expected, higher unemployment compensation causes the worker to hold out for higher wages

In effect, the cost of continuing job search is reduced

(Code to reproduce the figure can be found in [this directory](#))

**The Reservation Wage and Discounting** Next let's investigate how  $\bar{w}$  varies with the discount rate

The next figure plots the reservation wage associated with different values of  $\beta$



Again, the results are intuitive: More patient workers will hold out for higher wages

**The Reservation Wage and Job Destruction** Finally, let's look at how  $\bar{w}$  varies with the job separation rate  $\alpha$

Higher  $\alpha$  translates to a greater chance that a worker will face termination in each period once employed

Once more, the results are in line with our intuition

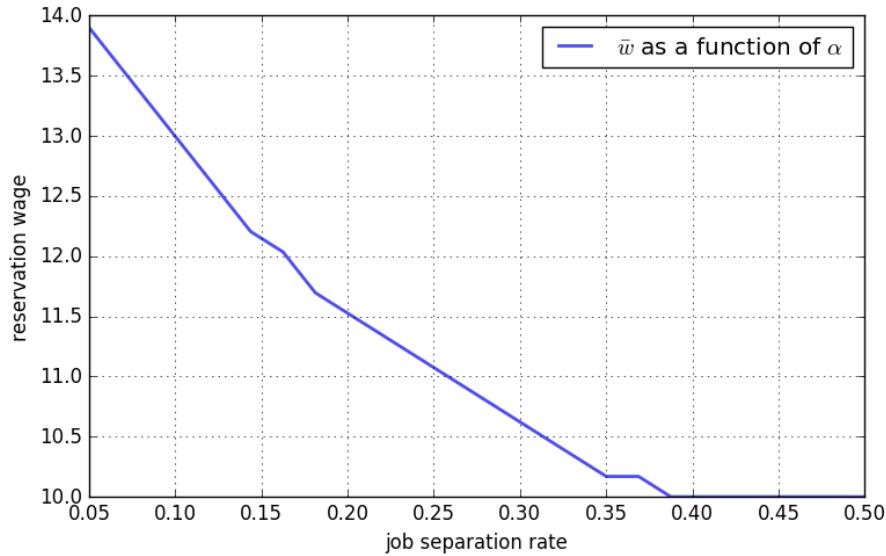
If the separation rate is high, then the benefit of holding out for a higher wage falls

Hence the reservation wage is lower

### Exercises

**Exercise 1** In the preceding discussion we computed the reservation wage for various instances of the McCall model

Try implementing your own function that accomplishes this task



Its input should be an instance of *McCallModel* as defined in `mccall_bellman_iteration.py` and its output should be the corresponding reservation wage

In doing so, you can make use of

- the logic for computing the reservation wage discussed above
- the code for computing value functions in `mccall_bellman_iteration.py`

**Exercise 2** Use your function from Exercise 1 to plot  $\bar{w}$  against the job offer rate  $\gamma$

Interpret your results

### Solutions

[Solution notebook](#)

## 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

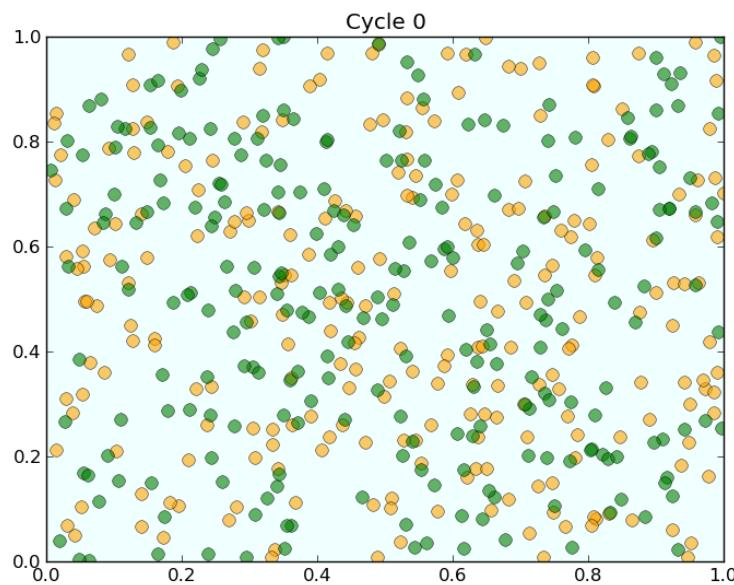
In this way, we cycle continuously through the agents, moving as required

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

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

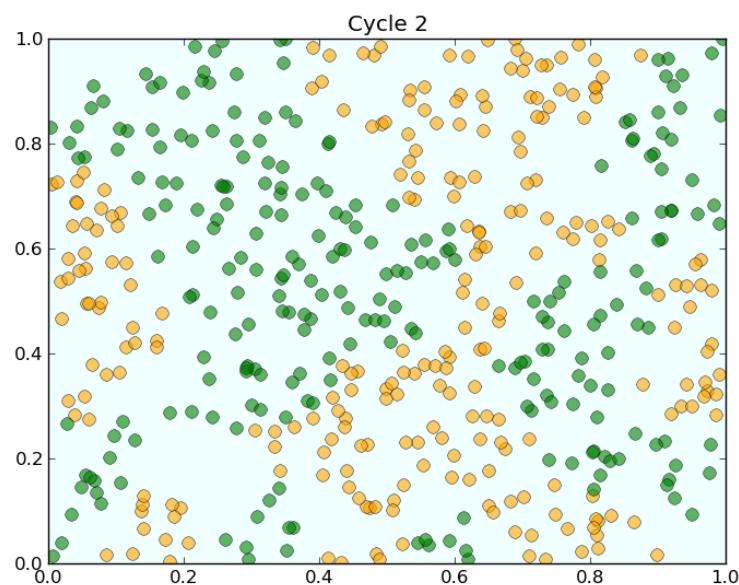
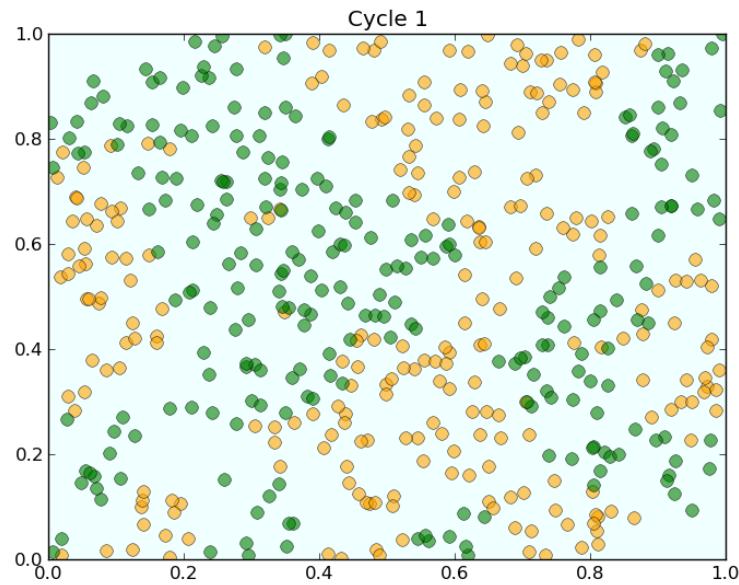
This is despite the fact that people in the model don't actually mind living mixed with the other type

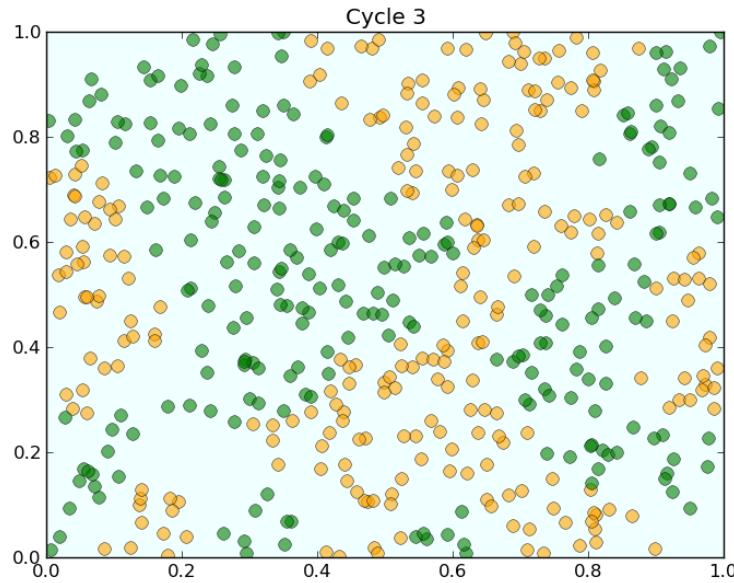
Even with these preferences, the outcome is a high degree of segregation

### Exercises

Rather than show you the program that generated these figures, we'll now ask you to write your own version

You can see our program at the end, when you look at the solution





**Exercise 1** Implement and run this simulation for yourself

Consider the following structure for your program

Agents are modeled as objects

(Have a look at *this lecture* if you've forgotten how to build your own 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

[Solution notebook](#)

# 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  $X_1, \dots, X_n$  be independent and identically distributed scalar random variables, with common distribution  $F$

When it exists, let  $\mu$  denote the common mean of this sample:

$$\mu := \mathbb{E}X = \int xF(dx)$$

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

$$\mathbb{P}\{\bar{X}_n \rightarrow \mu \text{ as } n \rightarrow \infty\} = 1 \quad (2.27)$$

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  $\bar{X}_n \rightarrow \mu$  can be evaluated

In this setting, (2.27) should be interpreted as meaning that the probability of the computer producing a sequence where  $\bar{X}_n \rightarrow \mu$  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  $X_1, \dots, X_n$  is IID with  $\mathbb{E}X_i^2 < \infty$ , then, for any  $\epsilon > 0$ , we have

$$\mathbb{P}\{|\bar{X}_n - \mu| \geq \epsilon\} \rightarrow 0 \quad \text{as } n \rightarrow \infty \quad (2.28)$$

(This version is weaker because we claim only **convergence in probability** rather than **almost sure convergence**, and assume a finite second moment)

To see that this is so, fix  $\epsilon > 0$ , and let  $\sigma^2$  be the variance of each  $X_i$

Recall the [Chebyshev inequality](#), which tells us that

$$\mathbb{P}\{|\bar{X}_n - \mu| \geq \epsilon\} \leq \frac{\mathbb{E}[(\bar{X}_n - \mu)^2]}{\epsilon^2} \quad (2.29)$$

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 (2.29), we come to the estimate

$$\mathbb{P}\{|\bar{X}_n - \mu| \geq \epsilon\} \leq \frac{\sigma^2}{n\epsilon^2} \quad (2.30)$$

The claim in (2.28) is now clear

Of course, if the sequence  $X_1, \dots, X_n$  is correlated, then the cross-product terms  $\mathbb{E}(X_i - \mu)(X_j - \mu)$  are not necessarily zero

While this doesn't mean that the same line of argument is impossible, it does mean that if we want a similar result then the covariances should be "almost zero" for "most" of these terms

In a long sequence, this would be true if, for example,  $\mathbb{E}(X_i - \mu)(X_j - \mu)$  approached zero when the difference between  $i$  and  $j$  became large

In other words, the LLN can still work if the sequence  $X_1, \dots, X_n$  has a kind of "asymptotic independence", in the sense that correlation falls to zero as variables become further apart in the sequence

This idea is very important in time series analysis, and we'll come across it again soon enough

**Illustration** Let's now illustrate the classical IID law of large numbers using simulation

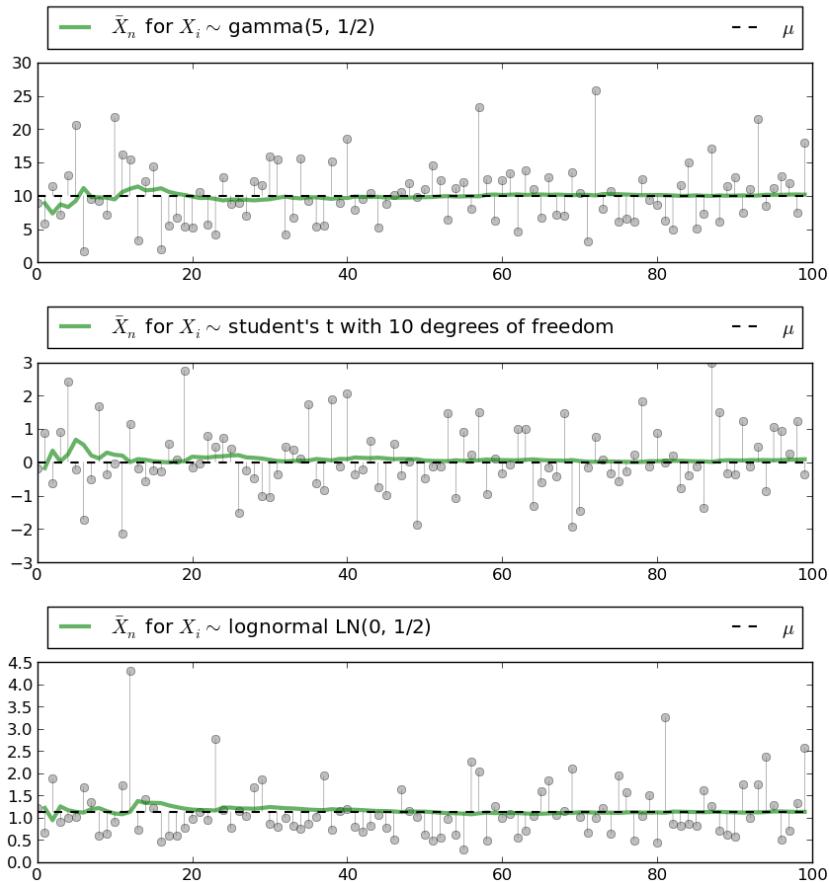
In particular, we aim to generate some sequences of IID random variables and plot the evolution of  $\bar{X}_n$  as  $n$  increases

Below is a figure that does just this (as usual, you can click on it to expand it)

It shows IID observations from three different distributions and plots  $\bar{X}_n$  against  $n$  in each case

The dots represent the underlying observations  $X_i$  for  $i = 1, \dots, 100$

In each of the three cases, convergence of  $\bar{X}_n$  to  $\mu$  occurs as predicted



The figure was produced by `illustates_lln.py`, which is shown below (and can be found in the `lln_clt` directory of the [applications repository](#))

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)
    xlabel = 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()

```

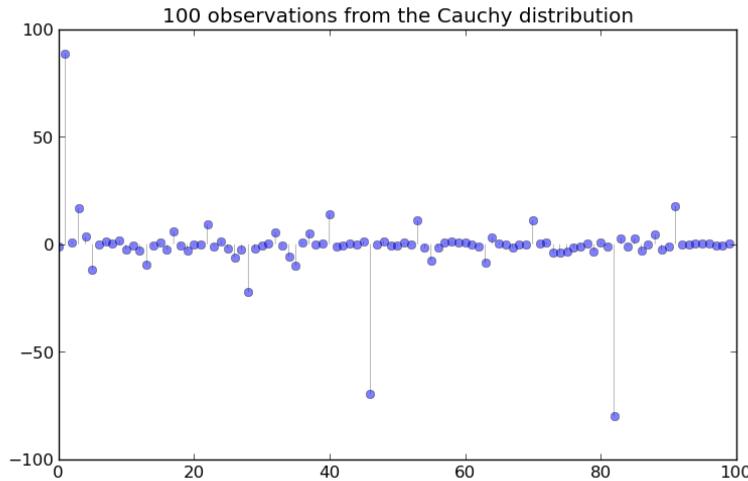
**Infinite Mean** What happens if the condition  $\mathbb{E}|X| < \infty$  in the statement of the LLN is not satisfied?

This might be the case if the underlying distribution is heavy tailed — the best known example is

the Cauchy distribution, which has density

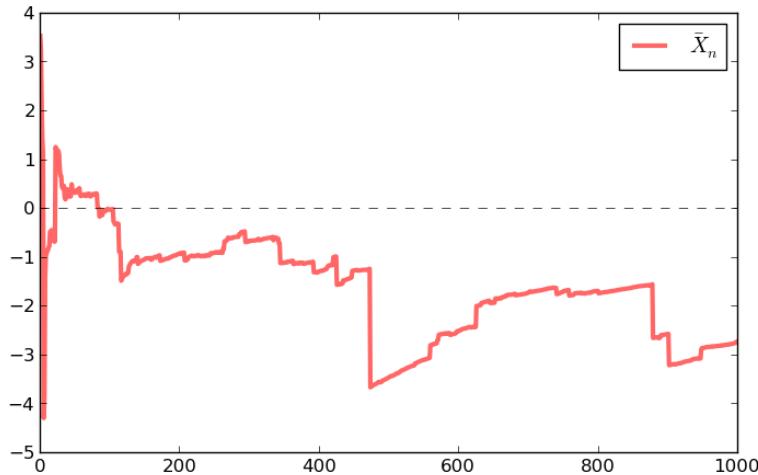
$$f(x) = \frac{1}{\pi(1+x^2)} \quad (x \in \mathbb{R})$$

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

$$\phi(t) = \mathbb{E}e^{itX} = \int e^{itx}f(x)dx = e^{-|t|} \quad (2.31)$$

Using independence, the characteristic function of the sample mean becomes

$$\begin{aligned}\mathbb{E}e^{it\bar{X}_n} &= \mathbb{E}\exp\left\{i\frac{t}{n}\sum_{j=1}^n X_j\right\} \\ &= \mathbb{E}\prod_{j=1}^n \exp\left\{i\frac{t}{n}X_j\right\} \\ &= \prod_{j=1}^n \mathbb{E}\exp\left\{i\frac{t}{n}X_j\right\} = [\phi(t/n)]^n\end{aligned}$$

In view of (2.31), 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  $X_1, \dots, X_n$  is IID, with common mean  $\mu$  and common variance  $\sigma^2 \in (0, \infty)$ , then

$$\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} N(0, \sigma^2) \quad \text{as } n \rightarrow \infty \quad (2.32)$$

Here  $\xrightarrow{d} N(0, \sigma^2)$  indicates convergence in distribution to a centered (i.e, zero mean) normal with standard deviation  $\sigma$

**Intuition** The striking implication of the CLT is that for **any** distribution with finite second moment, the simple operation of adding independent copies **always** leads to a Gaussian curve

A relatively simple proof of the central limit theorem can be obtained by working with characteristic functions (see, e.g., theorem 9.5.6 of [Dud02])

The proof is elegant but almost anticlimactic, and it provides surprisingly little intuition

In fact all of the proofs of the CLT that we know are similar in this respect

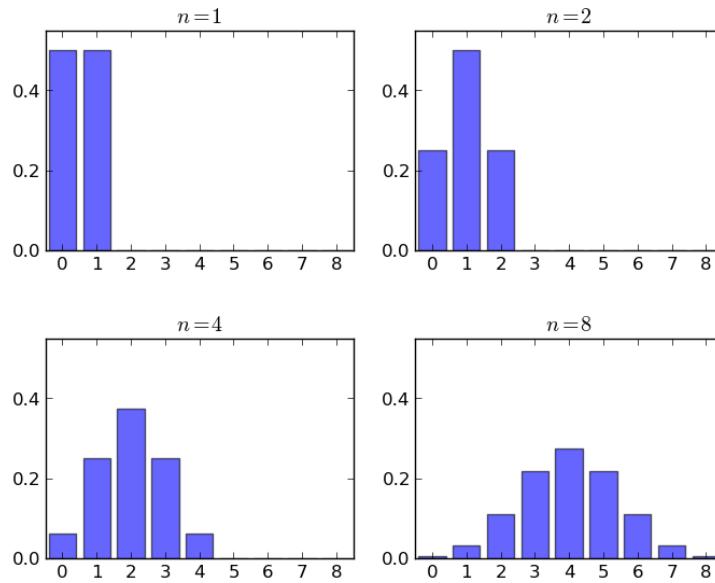
Why does adding independent copies produce a bell-shaped distribution?

Part of the answer can be obtained by investigating addition of independent Bernoulli random variables

In particular, let  $X_i$  be binary, with  $\mathbb{P}\{X_i = 0\} = \mathbb{P}\{X_i = 1\} = 0.5$ , and let  $X_1, \dots, X_n$  be independent

Think of  $X_i = 1$  as a “success”, so that  $Y_n = \sum_{i=1}^n X_i$  is the number of successes in  $n$  trials

The next figure plots the probability mass function of  $Y_n$  for  $n = 1, 2, 4, 8$



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()

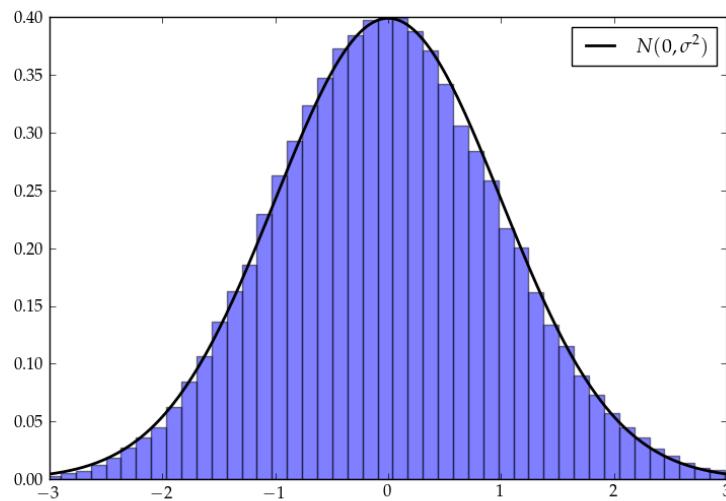
plt.show()

```

The file is `illustrates_clt.py`, from the [QuantEcon.applications](#) repo

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

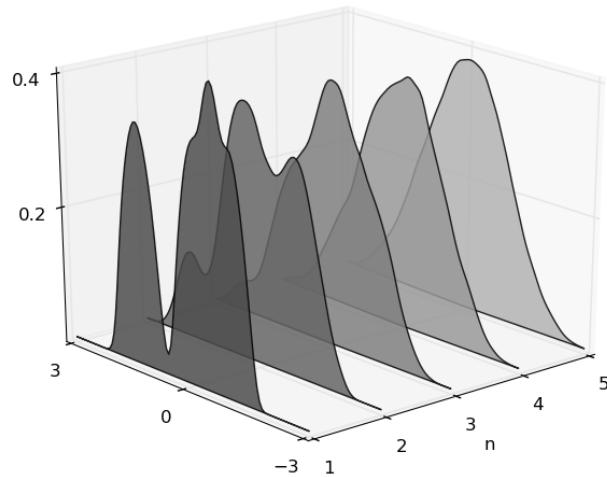
For  $n = 2$ , the distribution of  $Y_2$  is that of  $(X_1 + X_2)/\sqrt{2}$ , and so on

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 `lln_clt/clt3d.py`, which is available from the [applications repository](#)

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  $(X_1, \dots, X_k)$

Each realization of  $\mathbf{X}$  is an element of  $\mathbb{R}^k$

A collection of random vectors  $\mathbf{X}_1, \dots, \mathbf{X}_n$  is called independent if, given any  $n$  vectors  $\mathbf{x}_1, \dots, \mathbf{x}_n$  in  $\mathbb{R}^k$ , we have

$$\mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1, \dots, \mathbf{X}_n \leq \mathbf{x}_n\} = \mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1\} \times \dots \times \mathbb{P}\{\mathbf{X}_n \leq \mathbf{x}_n\}$$

(The vector inequality  $\mathbf{X} \leq \mathbf{x}$  means that  $X_j \leq x_j$  for  $j = 1, \dots, k$ )

Let  $\mu_j := \mathbb{E}[X_j]$  for all  $j = 1, \dots, k$

The expectation  $\mathbb{E}[\mathbf{X}]$  of  $\mathbf{X}$  is defined to be the vector of expectations:

$$\mathbb{E}[\mathbf{X}] := \begin{pmatrix} \mathbb{E}[X_1] \\ \mathbb{E}[X_2] \\ \vdots \\ \mathbb{E}[X_k] \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_k \end{pmatrix} =: \boldsymbol{\mu}$$

The *variance-covariance matrix* of random vector  $\mathbf{X}$  is defined as

$$\text{Var}[\mathbf{X}] := \mathbb{E}[(\mathbf{X} - \boldsymbol{\mu})(\mathbf{X} - \boldsymbol{\mu})']$$

Expanding this out, we get

$$\text{Var}[\mathbf{X}] = \begin{pmatrix} \mathbb{E}[(X_1 - \mu_1)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_1 - \mu_1)(X_k - \mu_k)] \\ \mathbb{E}[(X_2 - \mu_2)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_2 - \mu_2)(X_k - \mu_k)] \\ \vdots & \vdots & \vdots \\ \mathbb{E}[(X_k - \mu_k)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_k - \mu_k)(X_k - \mu_k)] \end{pmatrix}$$

The  $j, k$ -th term is the scalar covariance between  $X_j$  and  $X_k$

With this notation we can proceed to the multivariate LLN and CLT

Let  $\mathbf{X}_1, \dots, \mathbf{X}_n$  be a sequence of independent and identically distributed random vectors, each one taking values in  $\mathbb{R}^k$

Let  $\boldsymbol{\mu}$  be the vector  $\mathbb{E}[\mathbf{X}_i]$ , and let  $\Sigma$  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

$$\mathbb{P}\{\bar{\mathbf{X}}_n \rightarrow \boldsymbol{\mu} \text{ as } n \rightarrow \infty\} = 1 \quad (2.33)$$

Here  $\bar{\mathbf{X}}_n \rightarrow \boldsymbol{\mu}$  means that  $\|\bar{\mathbf{X}}_n - \boldsymbol{\mu}\| \rightarrow 0$ , where  $\|\cdot\|$  is the standard Euclidean norm

The CLT tells us that, provided  $\Sigma$  is finite,

$$\sqrt{n}(\bar{\mathbf{X}}_n - \boldsymbol{\mu}) \xrightarrow{d} N(\mathbf{0}, \Sigma) \quad \text{as } n \rightarrow \infty \quad (2.34)$$

### Exercises

**Exercise 1** One very useful consequence of the central limit theorem is as follows

Assume the conditions of the CLT as stated above

If  $g: \mathbb{R} \rightarrow \mathbb{R}$  is differentiable at  $\mu$  and  $g'(\mu) \neq 0$ , then

$$\sqrt{n}\{g(\bar{X}_n) - g(\mu)\} \xrightarrow{d} N(0, g'(\mu)^2\sigma^2) \quad \text{as } n \rightarrow \infty \quad (2.35)$$

This theorem is used frequently in statistics to obtain the asymptotic distribution of estimators — many of which can be expressed as functions of sample means

(These kinds of results are often said to use the “delta method”)

The proof is based on a Taylor expansion of  $g$  around the point  $\mu$

Taking the result as given, let the distribution  $F$  of each  $X_i$  be uniform on  $[0, \pi/2]$  and let  $g(x) = \sin(x)$

Derive the asymptotic distribution of  $\sqrt{n}\{g(\bar{X}_n) - g(\mu)\}$  and illustrate convergence in the same spirit as the program `illustrate_clt.py` discussed above

What happens when you replace  $[0, \pi/2]$  with  $[0, \pi]$ ?

What is the source of the problem?

**Exercise 2** Here’s a result that’s often used in developing statistical tests, and is connected to the multivariate central limit theorem

If you study econometric theory, you will see this result used again and again

Assume the setting of the multivariate CLT *discussed above*, so that

1.  $\mathbf{X}_1, \dots, \mathbf{X}_n$  is a sequence of IID random vectors, each taking values in  $\mathbb{R}^k$
2.  $\boldsymbol{\mu} := \mathbb{E}[\mathbf{X}_i]$ , and  $\Sigma$  is the variance-covariance matrix of  $\mathbf{X}_i$
3. The convergence

$$\sqrt{n}(\bar{\mathbf{X}}_n - \boldsymbol{\mu}) \xrightarrow{d} N(\mathbf{0}, \Sigma) \quad (2.36)$$

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

$$\text{Var}[\mathbf{AX}] = \mathbf{A} \text{Var}[\mathbf{X}] \mathbf{A}'$$

Second, by the **continuous mapping theorem**, if  $\mathbf{Z}_n \xrightarrow{d} \mathbf{Z}$  in  $\mathbb{R}^k$  and  $\mathbf{A}$  is constant and  $k \times k$ , then

$$\mathbf{AZ}_n \xrightarrow{d} \mathbf{AZ}$$

Third, if  $\mathbf{S}$  is a  $k \times k$  symmetric positive definite matrix, then there exists a symmetric positive definite matrix  $\mathbf{Q}$ , called the inverse **square root** of  $\mathbf{S}$ , such that

$$\mathbf{QSQ}' = \mathbf{I}$$

Here  $\mathbf{I}$  is the  $k \times k$  identity matrix

Putting these things together, your first exercise is to show that if  $\mathbf{Q}$  is the inverse square root of  $\Sigma$ , 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

$$n\|\mathbf{Q}(\bar{\mathbf{X}}_n - \boldsymbol{\mu})\|^2 \xrightarrow{d} \chi^2(k) \quad (2.37)$$

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 (2.37) 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  $\Sigma$  from the preceding information

## Solutions

[Solution notebook](#)

## 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  $t = 0, 1, 2, \dots$
- 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  $t = 0, 1, 2, \dots$
- An  $n \times n$  matrix  $A$  called the **transition matrix**
- An  $n \times m$  matrix  $C$  called the **volatility matrix**
- A  $k \times n$  matrix  $G$  sometimes called the **output matrix**

Here is the linear state-space system

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t \\ x_0 &\sim N(\mu_0, \Sigma_0) \end{aligned} \tag{2.38}$$

**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  $w_1, w_2, \dots$ , the model (2.38) 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 go through 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

$$\mathbb{E}[w_{t+1}|x_t, x_{t-1}, \dots] = 0$$

This is a weaker condition than that  $\{w_t\}$  is iid with  $w_{t+1} \sim N(0, I)$

**Examples** By appropriate choice of the primitives, a variety of dynamics can be represented in terms of the linear state space model

The following examples help to highlight this point

They also illustrate the wise dictum *finding the state is an art*

**Second-order difference equation** Let  $\{y_t\}$  be a deterministic sequence that satisfies

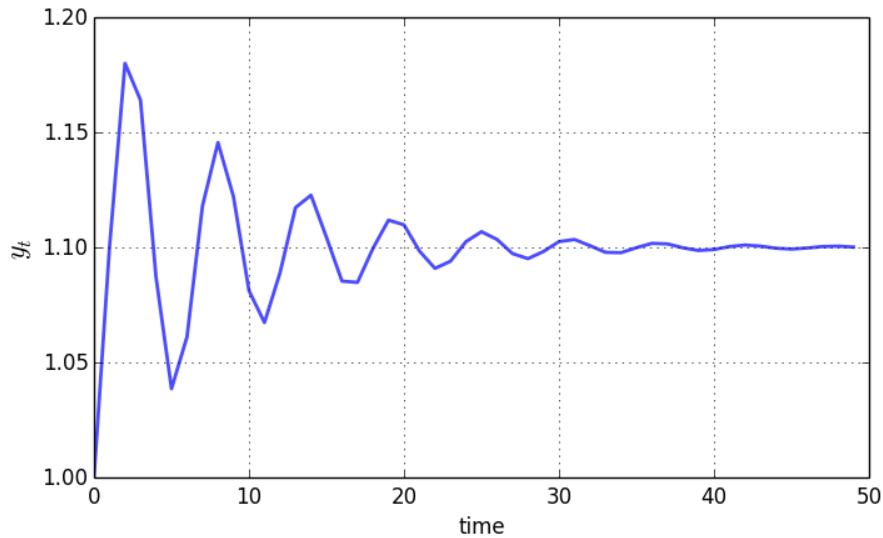
$$y_{t+1} = \phi_0 + \phi_1 y_t + \phi_2 y_{t-1} \quad \text{s.t. } y_0, y_{-1} \text{ given} \quad (2.39)$$

To map (2.39) into our state space system (2.38), we set

$$x_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix} \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \phi_0 & \phi_1 & \phi_2 \\ 0 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad G = [0 \ 1 \ 0]$$

You can confirm that under these definitions, (2.38) and (2.39) 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 (2.38) to represent the model

$$y_{t+1} = \phi_1 y_t + \phi_2 y_{t-1} + \phi_3 y_{t-2} + \phi_4 y_{t-3} + \sigma w_{t+1} \quad (2.40)$$

where  $\{w_t\}$  is iid and standard normal

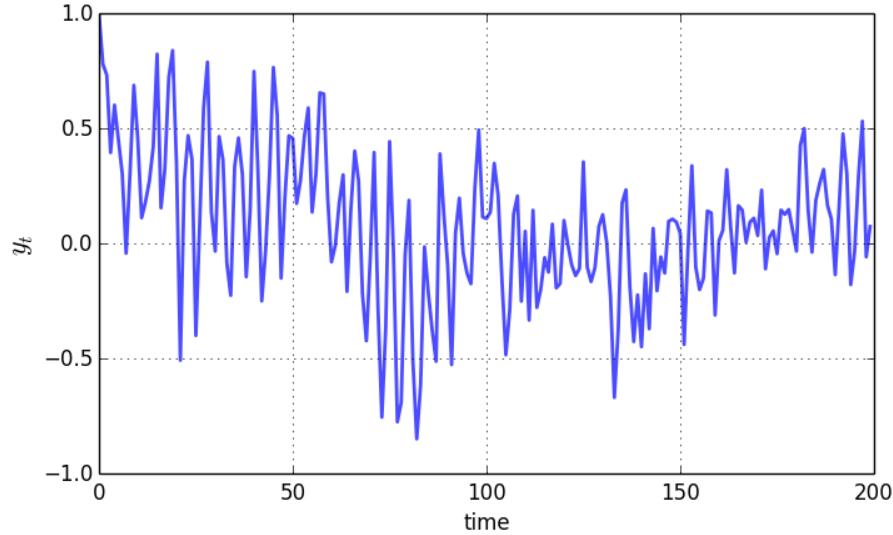
To put this in the linear state space format we take  $x_t = [y_t \ y_{t-1} \ y_{t-2} \ y_{t-3}]'$  and

$$A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad G = [1 \ 0 \ 0 \ 0]$$

The matrix  $A$  has the form of the *companion matrix* to the vector  $[\phi_1 \ \phi_2 \ \phi_3 \ \phi_4]$ .

The next figure shows dynamics of this process when

$$\phi_1 = 0.5, \phi_2 = -0.2, \phi_3 = 0, \phi_4 = 0.5, \sigma = 0.2, y_0 = y_{-1} = y_{-2} = y_{-3} = 1$$



**Vector Autoregressions** Now suppose that

- $y_t$  is a  $k \times 1$  vector
- $\phi_j$  is a  $k \times k$  matrix and
- $w_t$  is  $k \times 1$

Then (2.40) is termed a *vector autoregression*

To map this into (2.38), we set

$$x_t = \begin{bmatrix} y_t \\ y_{t-1} \\ y_{t-2} \\ y_{t-3} \end{bmatrix} \quad A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \end{bmatrix} \quad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad G = [I \ 0 \ 0 \ 0]$$

where  $I$  is the  $k \times k$  identity matrix and  $\sigma$  is a  $k \times k$  matrix

**Seasonals** We can use (2.38) 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 (2.40)

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:<sup>1</sup>

$$x_{t+4} = x_t$$

Such an  $x_t$  process can be used to model deterministic seasonals in quarterly time series.

The *indeterministic* seasonal produces recurrent, but aperiodic, seasonal fluctuations.

**Time Trends** The model  $y_t = at + b$  is known as a *linear time trend*

We can represent this model in the linear state space form by taking

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad G = [a \ b] \quad (2.41)$$

and starting at initial condition  $x_0 = [0 \ 1]'$

In fact it's possible to use the state-space system to represent polynomial trends of any order

For instance, let

$$x_0 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

It follows that

$$A^t = \begin{bmatrix} 1 & t & t(t-1)/2 \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix}$$

Then  $x'_t = [t(t-1)/2 \ t \ 1]$ , so that  $x_t$  contains linear and quadratic time trends

As a variation on the linear time trend model, consider  $y_t = at + b$

To modify (2.41) accordingly, we set

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad G = [a \ b] \quad (2.42)$$

---

<sup>1</sup> The eigenvalues of  $A$  are  $(1, -1, i, -i)$ .

**Moving Average Representations** A nonrecursive expression for  $x_t$  as a function of  $x_0, w_1, w_2, \dots, w_t$  can be found by using (2.38) repeatedly to obtain

$$\begin{aligned} x_t &= Ax_{t-1} + Cw_t \\ &= A^2x_{t-2} + ACw_{t-1} + Cw_t \\ &\quad \vdots \\ &= \sum_{j=0}^{t-1} A^j C w_{t-j} + A^t x_0 \end{aligned} \tag{2.43}$$

Representation (2.43) is a *moving average* representation

It expresses  $\{x_t\}$  as a linear function of

1. current and past values of the process  $\{w_t\}$  and
2. the initial condition  $x_0$

As an example of a moving average representation, let the model be

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

You will be able to show that  $A^t = \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix}$  and  $A^j C = [1 \ 0]'$

Substituting into the moving average representation (2.43), we obtain

$$x_{1t} = \sum_{j=0}^{t-1} w_{t-j} + [1 \ t] x_0$$

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 (2.38), 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

$$\mu_{t+1} = A\mu_t \quad \text{with } \mu_0 \text{ given} \tag{2.44}$$

Here  $\mu_0$  is a primitive given in (2.38)

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

$$\Sigma_{t+1} = A\Sigma_t A' + CC' \quad \text{with } \Sigma_0 \text{ given} \quad (2.45)$$

As with  $\mu_0$ , the matrix  $\Sigma_0$  is a primitive given in (2.38)

As a matter of terminology, we will sometimes call

- $\mu_t$  the *unconditional mean* of  $x_t$
- $\Sigma_t$  the *unconditional variance-covariance matrix* of  $x_t$

This is to distinguish  $\mu_t$  and  $\Sigma_t$  from related objects that use conditioning information, to be defined below

However, you should be aware that these “unconditional” moments do depend on the initial distribution  $N(\mu_0, \Sigma_0)$

**Moments of the Observations** Using linearity of expectations again we have

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t] = G\mu_t \quad (2.46)$$

The variance-covariance matrix of  $y_t$  is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t] = G\Sigma_t G' \quad (2.47)$$

**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

One such situation is when the vector in question is Gaussian (i.e., normally distributed)

This is the case here, given

1. our Gaussian assumptions on the primitives
2. the fact that normality is preserved under linear operations

In fact, it's well-known that

$$u \sim N(\bar{u}, S) \quad \text{and} \quad v = a + Bu \implies v \sim N(a + B\bar{u}, BSB') \quad (2.48)$$

In particular, given our Gaussian assumptions on the primitives and the linearity of (2.38) we can see immediately that both  $x_t$  and  $y_t$  are Gaussian for all  $t \geq 0$ <sup>2</sup>

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 (2.44) and (2.45)

---

<sup>2</sup> The correct way to argue this is by induction. Suppose that  $x_t$  is Gaussian. Then (2.38) and (2.48) 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.

Letting  $\mu_t$  and  $\Sigma_t$  be as defined by these equations, we have

$$x_t \sim N(\mu_t, \Sigma_t) \quad (2.49)$$

By similar reasoning combined with (2.46) and (2.47),

$$y_t \sim N(G\mu_t, G\Sigma_t G') \quad (2.50)$$

**Ensemble Interpretations** How should we interpret the distributions defined by (2.49)–(2.50)?

Intuitively, the probabilities in a distribution correspond to relative frequencies in a large population drawn from that distribution

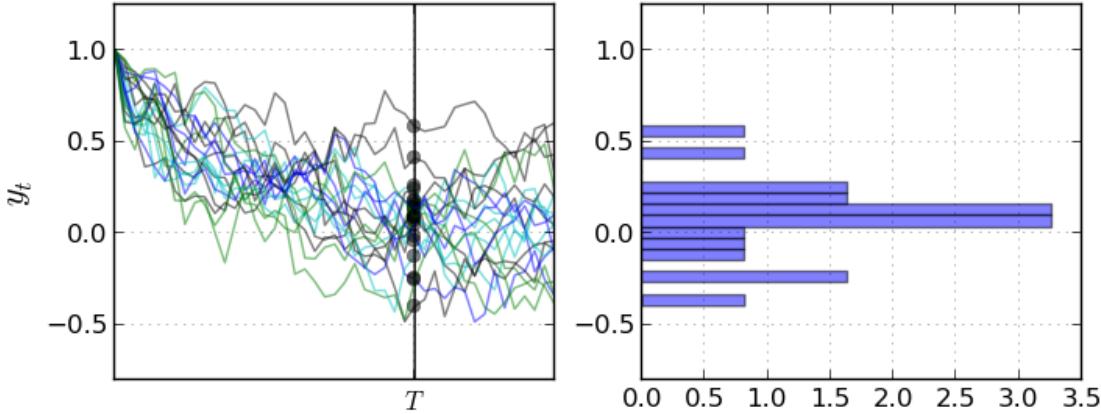
Let's apply this idea to our setting, focusing on the distribution of  $y_T$  for fixed  $T$

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 (2.40)

The values of  $y_T$  are represented by black dots in the left-hand figure



In the right-hand figure, these values are converted into a rotated histogram that shows relative frequencies from our sample of 20  $y_T$ 's

(The parameters and source code for the figures can be found in file `linear_models/paths_and_hist.py` from the [applications repository](#))

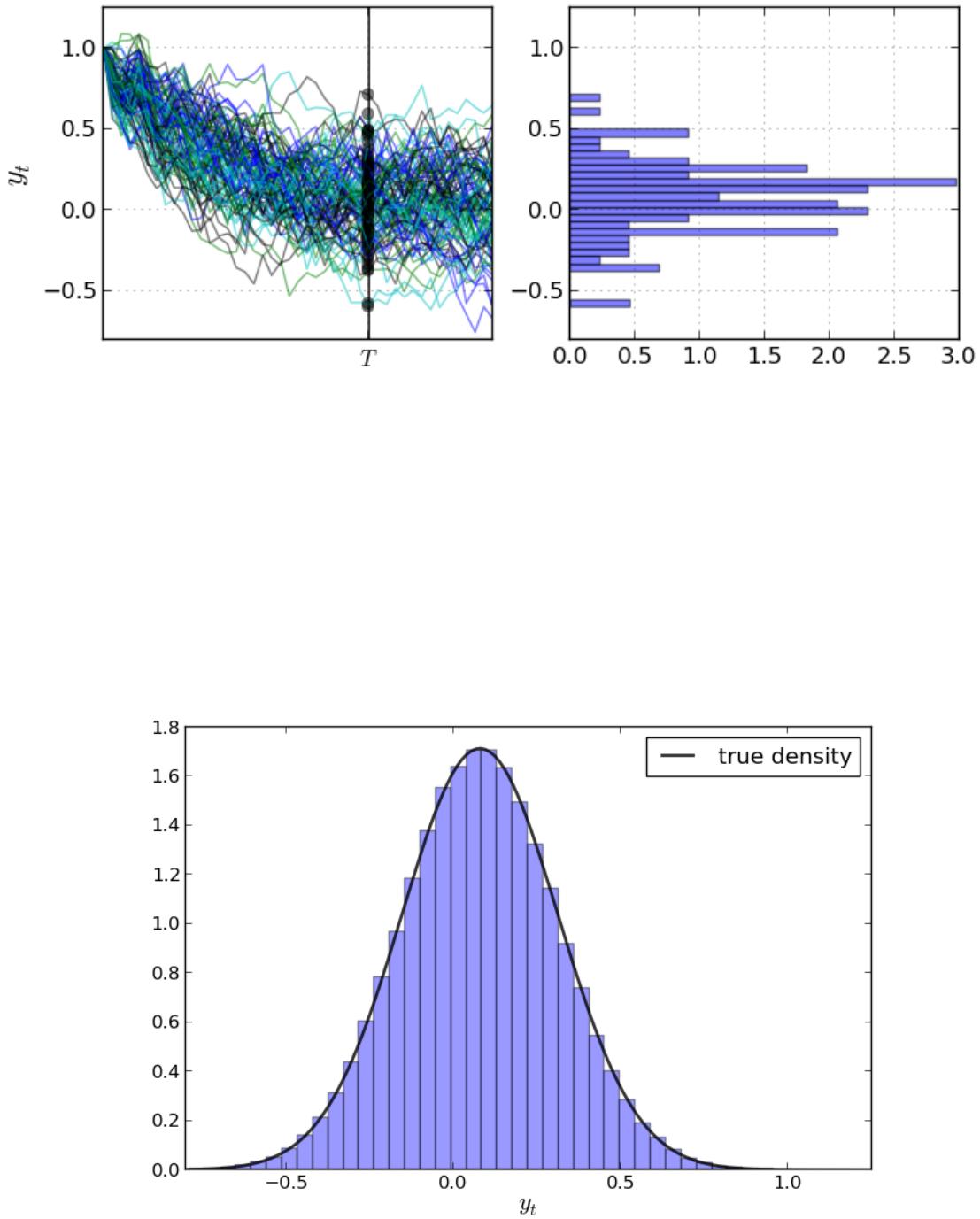
Here is another figure, this time with 100 observations

Let's now try with 500,000 observations, showing only the histogram (without rotation)

The black line is the density of  $y_T$  calculated analytically, using (2.50)

The histogram and analytical distribution are close, as expected

By looking at the figures and experimenting with parameters, you will gain a feel for how the distribution depends on the model primitives *listed above*



**Ensemble means** In the preceding figure we recovered the 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

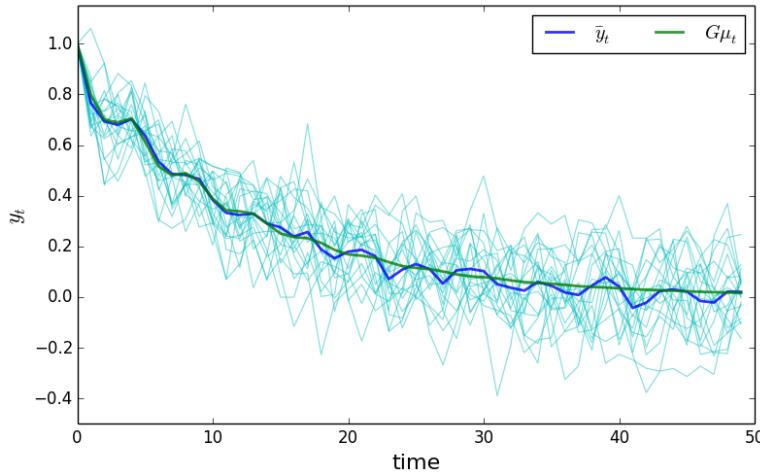
Just as the histogram corresponds to the distribution, the *ensemble* or *cross-sectional average*

$$\bar{y}_T := \frac{1}{I} \sum_{i=1}^I y_T^i$$

approximates the expectation  $\mathbb{E}[y_T] = G\mu_T$  (as implied by the law of large numbers)

Here's a simulation comparing the ensemble averages and population means at time points  $t = 0, \dots, 50$

The parameters are the same as for the preceding figures, and the sample size is relatively small ( $I = 20$ )



The ensemble mean for  $x_t$  is

$$\bar{x}_T := \frac{1}{I} \sum_{i=1}^I x_T^i \rightarrow \mu_T \quad (I \rightarrow \infty)$$

The limit  $\mu_T$  can be thought of as a “population average”

(By *population 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

$$\frac{1}{I} \sum_{i=1}^I (x_T^i - \bar{x}_T)(x_T^i - \bar{x}_T)' \rightarrow \Sigma_T \quad (I \rightarrow \infty)$$

**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  $x_0, x_1, \dots, x_T$ , recall that joint and conditional densities are linked by the rule

$$p(x, y) = p(y | x)p(x) \quad (\text{joint} = \text{conditional} \times \text{marginal})$$

From this rule we get  $p(x_0, x_1) = p(x_1 | x_0)p(x_0)$

The Markov property  $p(x_t | x_{t-1}, \dots, x_0) = p(x_t | x_{t-1})$  and repeated applications of the preceding rule lead us to

$$p(x_0, x_1, \dots, x_T) = p(x_0) \prod_{t=0}^{T-1} p(x_{t+1} | x_t)$$

The marginal  $p(x_0)$  is just the primitive  $N(\mu_0, \Sigma_0)$

In view of (2.38), the conditional densities are

$$p(x_{t+1} | x_t) = N(Ax_t, CC')$$

**Autocovariance functions** An important object related to the joint distribution is the *autocovariance function*

$$\Sigma_{t+j,t} := \mathbb{E} [(x_{t+j} - \mu_{t+j})(x_t - \mu_t)'] \quad (2.51)$$

Elementary calculations show that

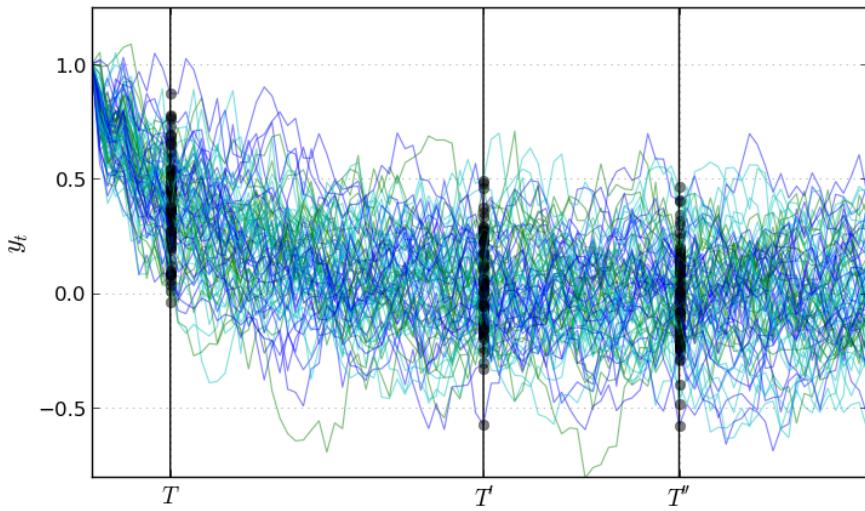
$$\Sigma_{t+j,t} = A^j \Sigma_t \quad (2.52)$$

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  $y_t$  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  $t \rightarrow \infty$

When such a distribution exists it is called a *stationary distribution*

**Stationary Distributions** In our setting, a distribution  $\psi_\infty$  is said to be *stationary* for  $x_t$  if

$$x_t \sim \psi_\infty \quad \text{and} \quad x_{t+1} = Ax_t + Cw_{t+1} \implies x_{t+1} \sim \psi_\infty$$

Since

1. in the present case all distributions are Gaussian
2. a Gaussian distribution is pinned down by its mean and variance-covariance matrix

we can restate the definition as follows:  $\psi_\infty$  is stationary for  $x_t$  if

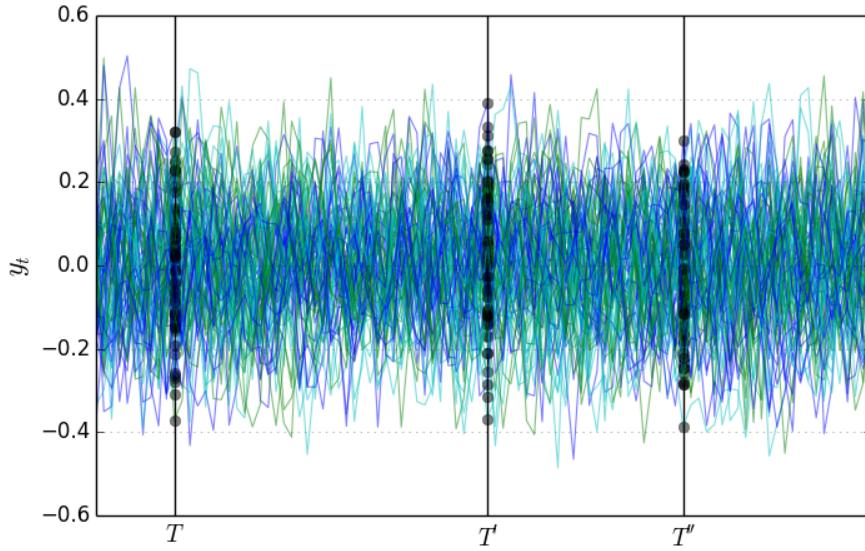
$$\psi_\infty = N(\mu_\infty, \Sigma_\infty)$$

where  $\mu_\infty$  and  $\Sigma_\infty$  are fixed points of (2.44) and (2.45) respectively

**Covariance Stationary Processes** Let's see what happens to the preceding figure if we start  $x_0$  at the stationary distribution

Now the differences in the observed distributions at  $T, T'$  and  $T''$  come entirely from random fluctuations due to the finite sample size

By



- our choosing  $x_0 \sim N(\mu_\infty, \Sigma_\infty)$
- the definitions of  $\mu_\infty$  and  $\Sigma_\infty$  as fixed points of (2.44) and (2.45) 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 (2.52), the autocovariance function takes the form  $\Sigma_{t+j,t} = A^j \Sigma_\infty$ , which depends on  $j$  but not on  $t$

This motivates the following definition

A process  $\{x_t\}$  is said to be *covariance stationary* if

- both  $\mu_t$  and  $\Sigma_t$  are constant in  $t$
- $\Sigma_{t+j,t}$  depends on the time gap  $j$  but not on time  $t$

In our setting,  $\{x_t\}$  will be covariance stationary if  $\mu_0, \Sigma_0, A, C$  assume values that imply that none of  $\mu_t, \Sigma_t, \Sigma_{t+j,t}$  depends on  $t$

### Conditions for Stationarity

**The globally stable case** The difference equation  $\mu_{t+1} = A\mu_t$  is known to have *unique* fixed point  $\mu_\infty = 0$  if all eigenvalues of  $A$  have moduli strictly less than unity

That is, if `(np.abs(np.linalg.eigvals(A)) < 1).all() == True`

The difference equation (2.45) also has a unique fixed point in this case, and, moreover

$$\mu_t \rightarrow \mu_\infty = 0 \quad \text{and} \quad \Sigma_t \rightarrow \Sigma_\infty \quad \text{as} \quad t \rightarrow \infty$$

regardless of the initial conditions  $\mu_0$  and  $\Sigma_0$

This is the *globally stable case* — see these notes for more a theoretical treatment

However, global stability is more than we need for stationary solutions, and often more than we want

To illustrate, consider *our second order difference equation example*

Here the state is  $x_t = [1 \ y_t \ y_{t-1}]'$

Because of the constant first component in the state vector, we will never have  $\mu_t \rightarrow 0$

How can we find stationary solutions that respect a constant state component?

**Processes with a constant state component** To investigate such a process, suppose that  $A$  and  $C$  take the form

$$A = \begin{bmatrix} A_1 & a \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} C_1 \\ 0 \end{bmatrix}$$

where

- $A_1$  is an  $(n - 1) \times (n - 1)$  matrix
- $a$  is an  $(n - 1) \times 1$  column vector

Let  $x_t = [x'_{1t} \ 1]'$  where  $x_{1t}$  is  $(n - 1) \times 1$

It follows that

$$x_{1,t+1} = A_1 x_{1t} + a + C_1 w_{t+1}$$

Let  $\mu_{1t} = \mathbb{E}[x_{1t}]$  and take expectations on both sides of this expression to get

$$\mu_{1,t+1} = A_1 \mu_{1,t} + a \tag{2.53}$$

Assume now that the moduli of the eigenvalues of  $A_1$  are all strictly less than one

Then (2.53) has a unique stationary solution, namely,

$$\mu_{1\infty} = (I - A_1)^{-1}a$$

The stationary value of  $\mu_t$  itself is then  $\mu_\infty := [\mu'_{1\infty} \ 1]'$

The stationary values of  $\Sigma_t$  and  $\Sigma_{t+j,t}$  satisfy

$$\begin{aligned} \Sigma_\infty &= A \Sigma_\infty A' + CC' \\ \Sigma_{t+j,t} &= A^j \Sigma_\infty \end{aligned} \tag{2.54}$$

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 (2.45) converge to the fixed point of the *discrete Lyapunov equation* in the first line of (2.54)

---

**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  $\mu_\infty$  as the sample size  $T$  approaches infinity

**Averages over time** Ensemble averages across simulations are interesting theoretically, but in real life we usually observe only a *single* realization  $\{x_t, y_t\}_{t=0}^T$

So now let's take a single realization and form the time series averages

$$\bar{x} := \frac{1}{T} \sum_{t=1}^T x_t \quad \text{and} \quad \bar{y} := \frac{1}{T} \sum_{t=1}^T y_t$$

Do these time series averages converge to something interpretable in terms of our basic state-space representation?

The answer depends on something called *ergodicity*

Ergodicity is the property that time series and ensemble averages coincide

More formally, ergodicity implies that time series sample averages converge to their expectation under the stationary distribution

In particular,

- $\frac{1}{T} \sum_{t=1}^T x_t \rightarrow \mu_\infty$
- $\frac{1}{T} \sum_{t=1}^T (x_t - \bar{x}_T)(x_t - \bar{x}_T)' \rightarrow \Sigma_\infty$
- $\frac{1}{T} \sum_{t=1}^T (x_{t+j} - \bar{x}_T)(x_t - \bar{x}_T)' \rightarrow A^j \Sigma_\infty$

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

$$\begin{aligned}x_{t+1} &= Ax_t + Cw_{t+1} \\y_t &= Gx_t + Hv_t \\x_0 &\sim N(\mu_0, \Sigma_0)\end{aligned}\tag{2.55}$$

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 (2.46) and (2.47) now become

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t + Hv_t] = G\mu_t\tag{2.56}$$

The variance-covariance matrix of  $y_t$  is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t + Hv_t] = G\Sigma_t G' + HH'\tag{2.57}$$

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

$$\mathbb{E}_t[x_{t+1}] := \mathbb{E}[x_{t+1} | x_t, x_{t-1}, \dots, x_0] = Ax_t$$

The right-hand side follows from  $x_{t+1} = Ax_t + Cw_{t+1}$  and the fact that  $w_{t+1}$  is zero mean and independent of  $x_t, x_{t-1}, \dots, x_0$

That  $\mathbb{E}_t[x_{t+1}] = \mathbb{E}[x_{t+1} | x_t]$  is an implication of  $\{x_t\}$  having the *Markov property*

The one-step-ahead forecast error is

$$x_{t+1} - \mathbb{E}_t[x_{t+1}] = Cw_{t+1}$$

The covariance matrix of the forecast error is

$$\mathbb{E}[(x_{t+1} - \mathbb{E}_t[x_{t+1}])(x_{t+1} - \mathbb{E}_t[x_{t+1}])'] = CC'$$

More generally, we'd like to compute the  $j$ -step ahead forecasts  $\mathbb{E}_t[x_{t+j}]$  and  $\mathbb{E}_t[y_{t+j}]$

With a bit of algebra we obtain

$$x_{t+j} = A^j x_t + A^{j-1} C w_{t+1} + A^{j-2} C w_{t+2} + \cdots + A^0 C w_{t+j}$$

In view of the iid property, current and past state values provide no information about future values of the shock

$$\text{Hence } \mathbb{E}_t[w_{t+k}] = \mathbb{E}[w_{t+k}] = 0$$

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

$$x_{t+j} - \mathbb{E}_t[x_{t+j}] = \sum_{s=0}^{j-1} A^s C w_{t-s+j} \quad (2.58)$$

Evidently,

$$V_j := \mathbb{E}_t[(x_{t+j} - \mathbb{E}_t[x_{t+j}]) (x_{t+j} - \mathbb{E}_t[x_{t+j}])'] = \sum_{k=0}^{j-1} A^k C C' A^k' \quad (2.59)$$

$V_j$  defined in (2.59) can be calculated recursively via  $V_1 = CC'$  and

$$V_j = CC' + AV_{j-1}A', \quad j \geq 2 \quad (2.60)$$

$V_j$  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

$$V_\infty = CC' + AV_\infty A' \quad (2.61)$$

Equation (2.61) is an example of a *discrete Lyapunov equation* in the covariance matrix  $V_\infty$

A sufficient condition for  $V_j$  to converge is that the eigenvalues of  $A$  be strictly less than one in modulus.

Weaker sufficient conditions for convergence associate eigenvalues equaling or exceeding one in modulus with elements of  $C$  that equal 0

**Forecasts of Geometric Sums** In several contexts, we want to compute forecasts of geometric sums of future random variables governed by the linear state-space system (2.38)

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  $I + \beta A + \beta^2 A^2 + \dots = [I - \beta A]^{-1}$

This leads to our formulas:

- Forecast of a geometric sum of future  $x$ 's

$$\mathbb{E}_t \left[ \sum_{j=0}^{\infty} \beta^j x_{t+j} \right] = [I + \beta A + \beta^2 A^2 + \dots] x_t = [I - \beta A]^{-1} x_t$$

- Forecast of a geometric sum of future  $y$ 's

$$\mathbb{E}_t \left[ \sum_{j=0}^{\infty} \beta^j y_{t+j} \right] = G[I + \beta A + \beta^2 A^2 + \dots] x_t = G[I - \beta A]^{-1} x_t$$

## 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.)

We repeat it here for convenience

```
"""
Filename: lss.py
Reference: http://quant-econ.net/py/linear_models.html

Computes quantities associated with the Gaussian linear state space model.
"""

from textwrap import dedent
import numpy as np
from numpy.random import multivariate_normal
from scipy.linalg import solve
from numba import jit

@jit
def simulate_linear_model(A, x0, v, ts_length):
    """
```

This is a separate function for simulating a vector linear system of the form

$$x_{t+1} = A x_t + v_t \text{ given } x_0 = x_0$$

Here  $x_t$  and  $v_t$  are both  $n \times 1$  and  $A$  is  $n \times n$ .

The purpose of separating this functionality out is to target it for optimization by Numba. For the same reason, matrix multiplication is broken down into for loops.

Parameters

-----

$A$  : array\_like or scalar(float)  
Should be  $n \times n$

$x_0$  : array\_like  
Should be  $n \times 1$ . Initial condition

$v$  : np.ndarray  
Should be  $n \times ts\_length-1$ . Its  $t$ -th column is used as the time  $t$  shock  $v_t$

$ts\_length$  : int  
The length of the time series

Returns

-----

$x$  : np.ndarray  
Time series with  $ts\_length$  columns, the  $t$ -th column being  $x_t$

"""

```
A = np.asarray(A)
n = A.shape[0]
x = np.empty((n, ts_length))
x[:, 0] = x0
for t in range(ts_length-1):
    #  $x[:, t+1] = A \cdot dot(x[:, t]) + v[:, t]$ 
    for i in range(n):
        x[i, t+1] = v[i, t]                      # Shock
        for j in range(n):
            x[i, t+1] += A[i, j] * x[j, t]      # Dot Product
return x
```

class LinearStateSpace(object):

"""

A class that describes a Gaussian linear state space model of the form:

$$x_{t+1} = A x_t + C w_{t+1}$$

$$y_t = G x_t + H v_t$$

where  $\{w_t\}$  and  $\{v_t\}$  are independent and standard normal with dimensions  $k$  and  $l$  respectively. The initial conditions are  $\mu_0$  and  $\Sigma_0$  for  $x_0 \sim N(\mu_0, \Sigma_0)$ . When  $\Sigma_0=0$ , the draw of  $x_0$  is exactly  $\mu_0$ .

```

Parameters
-----
A : array_like or scalar(float)
    Part of the state transition equation. It should be `n x n`
C : array_like or scalar(float)
    Part of the state transition equation. It should be `n x m`
G : array_like or scalar(float)
    Part of the observation equation. It should be `k x n`
H : array_like or scalar(float), optional(default=None)
    Part of the observation equation. It should be `k x l`
mu_0 : array_like or scalar(float), optional(default=None)
    This is the mean of initial draw and is `n x 1`
Sigma_0 : array_like or scalar(float), optional(default=None)
    This is the variance of the initial draw and is `n x n` and
    also should be positive definite and symmetric

Attributes
-----
A, C, G, H, mu_0, Sigma_0 : see Parameters
n, k, m, l : scalar(int)
    The dimensions of x_t, y_t, w_t and v_t respectively

"""
def __init__(self, A, C, G, H=None, mu_0=None, Sigma_0=None):
    self.A, self.G, self.C = list(map(self.convert, (A, G, C)))
    #-Check Input Shapes-
    ni,nj = self.A.shape
    if ni != nj:
        raise ValueError("Matrix A (shape: %s) needs to be square" % (self.A.shape))
    if ni != self.C.shape[0]:
        raise ValueError("Matrix C (shape: %s) does not have compatible dimensions with A. It should be (%s, %s)" % (self.C.shape, ni, nj))
    self.m = self.C.shape[1]
    self.k, self.n = self.G.shape
    if self.n != ni:
        raise ValueError("Matrix G (shape: %s) does not have compatible dimensions with A (%s)" % (self.G.shape, A.shape))
    if H is None:
        self.H = None
        self.l = None
    else:
        self.H = self.convert(H)
        self.l = self.H.shape[1]
    if mu_0 is None:
        self.mu_0 = np.zeros((self.n, 1))
    else:
        self.mu_0 = self.convert(mu_0)
        self.mu_0.shape = self.n, 1
    if Sigma_0 is None:
        self.Sigma_0 = np.zeros((self.n, self.n))
    else:
        self.Sigma_0 = self.convert(Sigma_0)

def __repr__(self):

```

```

    return self.__str__()

def __str__(self):
    m = """\
    Linear Gaussian state space model:
    - dimension of state space           : {n}
    - number of innovations             : {m}
    - dimension of observation equation : {k}
    """
    return dedent(m.format(n=self.n, k=self.k, m=self.m))

def convert(self, x):
    """
    Convert array_like objects (lists of lists, floats, etc.) into
    well formed 2D NumPy arrays

    """
    return np.atleast_2d(np.asarray(x, dtype='float'))

def simulate(self, ts_length=100):
    """
    Simulate a time series of length ts_length, first drawing

    x_0 ~ N(mu_0, Sigma_0)

    Parameters
    -----
    ts_length : scalar(int), optional(default=100)
        The length of the simulation

    Returns
    -----
    x : array_like(float)
        An n x ts_length array, where the t-th column is x_t
    y : array_like(float)
        A k x ts_length array, where the t-th column is y_t

    """
    x0 = multivariate_normal(self.mu_0.flatten(), self.Sigma_0)
    w = np.random.randn(self.m, ts_length-1)
    v = self.C.dot(w) # Multiply each w_t by C to get v_t = C w_t
    # == simulate time series == #
    x = simulate_linear_model(self.A, x0, v, ts_length)

    if self.H is not None:
        v = np.random.randn(self.l, ts_length)
        y = self.G.dot(x) + self.H.dot(v)
    else:
        y = self.G.dot(x)

    return x, y

```

```

def replicate(self, T=10, num_reps=100):
    """
    Simulate num_reps observations of x_T and y_T given
    x_0 ~ N(mu_0, Sigma_0).

    Parameters
    -----
    T : scalar(int), optional(default=10)
        The period that we want to replicate values for
    num_reps : scalar(int), optional(default=100)
        The number of replications that we want

    Returns
    -----
    x : array_like(float)
        An n x num_reps array, where the j-th column is the j-th
        observation of x_T

    y : array_like(float)
        A k x num_reps array, where the j-th column is the j-th
        observation of y_T

    """
    x = np.empty((self.n, num_reps))
    for j in range(num_reps):
        x_T, _ = self.simulate(ts_length=T+1)
        x[:, j] = x_T[:, -1]
    if self.H is not None:
        v = np.random.randn(self.l, num_reps)
        y = self.G.dot(x) + self.H.dot(v)
    else:
        y = self.G.dot(x)

    return x, y

def moment_sequence(self):
    """
    Create a generator to calculate the population mean and
    variance-covariance matrix for both x_t and y_t, starting at
    the initial condition (self.mu_0, self.Sigma_0). Each iteration
    produces a 4-tuple of items (mu_x, mu_y, Sigma_x, Sigma_y) for
    the next period.

    Yields
    -----
    mu_x : array_like(float)
        An n x 1 array representing the population mean of x_t
    mu_y : array_like(float)
        A k x 1 array representing the population mean of y_t
    Sigma_x : array_like(float)
        An n x n array representing the variance-covariance matrix
        of x_t
    Sigma_y : array_like(float)
    """

```

```

A k x k array representing the variance-covariance matrix
of y_t

"""
# == Simplify names == #
A, C, G, H = self.A, self.C, self.G, self.H
# == Initial moments == #
mu_x, Sigma_x = self.mu_0, self.Sigma_0

while 1:
    mu_y = G.dot(mu_x)
    if H is None:
        Sigma_y = G.dot(Sigma_x).dot(G.T)
    else:
        Sigma_y = G.dot(Sigma_x).dot(G.T) + H.dot(H.T)

    yield mu_x, mu_y, Sigma_x, Sigma_y

    # == Update moments of x == #
    mu_x = A.dot(mu_x)
    Sigma_x = A.dot(Sigma_x).dot(A.T) + C.dot(C.T)

def stationary_distributions(self, max_iter=200, tol=1e-5):
    """
    Compute the moments of the stationary distributions of x_t and
    y_t if possible. Computation is by iteration, starting from the
    initial conditions self.mu_0 and self.Sigma_0

    Parameters
    -----
    max_iter : scalar(int), optional(default=200)
        The maximum number of iterations allowed
    tol : scalar(float), optional(default=1e-5)
        The tolerance level that one wishes to achieve

    Returns
    -----
    mu_x_star : array_like(float)
        An n x 1 array representing the stationary mean of x_t
    mu_y_star : array_like(float)
        An k x 1 array representing the stationary mean of y_t
    Sigma_x_star : array_like(float)
        An n x n array representing the stationary var-cov matrix
        of x_t
    Sigma_y_star : array_like(float)
        An k x k array representing the stationary var-cov matrix
        of y_t

    """
    # == Initialize iteration == #
    m = self.moment_sequence()
    mu_x, mu_y, Sigma_x, Sigma_y = next(m)
    i = 0

```

```

    error = tol + 1

    # == Loop until convergence or failure == #
    while error > tol:

        if i > max_iter:
            fail_message = 'Convergence failed after {} iterations'
            raise ValueError(fail_message.format(max_iter))

        else:
            i += 1
            mu_x1, mu_y1, Sigma_x1, Sigma_y1 = next(m)
            error_mu = np.max(np.abs(mu_x1 - mu_x))
            error_Sigma = np.max(np.abs(Sigma_x1 - Sigma_x))
            error = max(error_mu, error_Sigma)
            mu_x, Sigma_x = mu_x1, Sigma_x1

    # == Prepare return values == #
    mu_x_star, Sigma_x_star = mu_x, Sigma_x
    mu_y_star, Sigma_y_star = mu_y1, Sigma_y1

    return mu_x_star, mu_y_star, Sigma_x_star, Sigma_y_star

def geometric_sums(self, beta, x_t):
    """
    Forecast the geometric sums

    
$$S_x := E [ \sum_{j=0}^{\infty} \beta^j x_{t+j} | x_t ]$$

    
$$S_y := E [ \sum_{j=0}^{\infty} \beta^j y_{t+j} | x_t ]$$


    Parameters
    -----
    beta : scalar(float)
        Discount factor, in [0, 1)

    beta : array_like(float)
        The term  $x_t$  for conditioning

    Returns
    -----
    S_x : array_like(float)
        Geometric sum as defined above

    S_y : array_like(float)
        Geometric sum as defined above

    """
    I = np.identity(self.n)
    S_x = solve(I - beta * self.A, x_t)
    S_y = self.G.dot(S_x)

    return S_x, S_y

```

```

def impulse_response(self, j=5):
    """
    Pulls off the impulse response coefficients to a shock
    in  $w_t$  for  $x$  and  $y$ 

    Important to note: We are uninterested in the shocks to
     $v$  for this method

    *  $x$  coefficients are  $C, AC, A^2 C\dots$ 
    *  $y$  coefficients are  $GC, GAC, GA^2C\dots$ 

    Parameters
    -----
    j : Scalar(int)
        Number of coefficients that we want

    Returns
    -----
    xcoef : list(array_like(float, 2))
        The coefficients for  $x$ 
    ycoef : list(array_like(float, 2))
        The coefficients for  $y$ 
    """
    # Pull out matrices
    A, C, G, H = self.A, self.C, self.G, self.H
    Apower = np.copy(A)

    # Create room for coefficients
    xcoef = [C]
    ycoef = [np.dot(G, C)]

    for i in range(j):
        xcoef.append(np.dot(Apower, C))
        ycoef.append(np.dot(G, np.dot(Apower, C)))
        Apower = np.dot(Apower, A)

    return xcoef, ycoef

```

Hopefully the code is relatively self explanatory and adequately documented

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

[Solution notebook](#)

# 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*

## 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:

- $\lambda$ , the job finding rate for currently unemployed workers
- $\alpha$ , the dismissal rate for currently employed workers
- $b$ , the entry rate into the labor force
- $d$ , the exit rate from the labor force

The growth rate of the labor force evidently equals  $g = b - d$

**Aggregate Variables** We want to derive the dynamics of the following aggregates

- $E_t$ , the total number of employed workers at date  $t$
- $U_t$ , the total number of unemployed workers at  $t$
- $N_t$ , the number of workers in the labor force at  $t$

We also want to know the values of the following objects

- The employment rate  $e_t := E_t / N_t$

- The unemployment rate  $u_t := U_t / N_t$

(Here and below, capital letters represent stocks and lowercase letters represent flows)

**Laws of Motion for Stock Variables** We begin by constructing laws of motion for the aggregate variables  $E_t, U_t, N_t$

Of the mass of workers  $E_t$  who are employed at date  $t$ ,

- $(1 - d)E_t$  will remain in the labor force
- of these,  $(1 - \alpha)(1 - d)E_t$  will remain employed

Of the mass of workers  $U_t$  workers who are currently unemployed,

- $(1 - d)U_t$  will remain in the labor force
- of these,  $(1 - d)\lambda U_t$  will become employed

Therefore, the number of workers who will be employed at date  $t + 1$  will be

$$E_{t+1} = (1 - d)(1 - \alpha)E_t + (1 - d)\lambda U_t$$

A similar analysis implies

$$U_{t+1} = (1 - d)\alpha E_t + (1 - d)(1 - \lambda)U_t + b(E_t + U_t)$$

The value  $b(E_t + U_t)$  is the mass of new workers entering the labor force unemployed

The total stock of workers  $N_t = E_t + U_t$  evolves as

$$N_{t+1} = (1 + b - d)N_t = (1 + g)N_t$$

Letting  $X_t := \begin{pmatrix} E_t \\ U_t \end{pmatrix}$ , 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  $\alpha, \lambda, b, d$
2. compute and store the implied objects  $g, A, \hat{A}$
3. provide methods to simulate dynamics of the stocks and rates
4. provide a method to compute the state state of the rate

To write an nice implementation, there's an issue we have to address

Derived data such as  $A$  depend on the primitives like  $\alpha$  and  $\lambda$

If a user alters these primitives, we would ideally like derived data to update automatically

(For example, if a user changes the value of  $b$  for a given instance of the class, we would like  $g = b - d$  to update automatically)

To achieve this outcome, we're going to use descriptors and decorators such as `@property`

If you need to refresh your understanding of how these work, consult [this lecture](#)

Here's the code, from the file `lake_model.py`

```
"""
Provides a class that simulates the dynamics of unemployment and employment in
the lake model.

"""

import numpy as np

class LakeModel:
    """
    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, alpha, b, d):
        self.lmda = lmda
        self.alpha = alpha
        self.b = b
        self.d = d
        self.g = b - d
        self.A = np.array([[1 - self.lmda, self.b], [self.d, 1 - self.alpha]])
        self.hat_A = np.linalg.inv(self.A)

    def simulate(self, x0, t):
        # Implement simulation logic here
        pass
```

```

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:`\mathbf{x}_{t+1} = \hat{\mathbf{A}} \mathbf{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 sequence of Employment and Unemployment stocks

    Parameters
    -----
    X0 : array
        Contains initial values (E0, U0)
    T : int
        Number of periods to simulate

    Returns
    -----
    X : iterator
        Contains sequence of employment and unemployment stocks
    """
    X = np.atleast_1d(X0) # recast as array just in case
    for t in range(T):
        yield X
        X = self.A @ X

def simulate_rate_path(self, x0, T):

```

```

r"""
Simulates the sequence of employment and unemployment rates.

Parameters
-----
x0 : array
    Contains initial values (e0,u0)
T : int
    Number of periods to simulate

Returns
-----
x : iterator
    Contains sequence of employment and unemployment rates

"""
x = np.atleast_1d(x0) # recast as array just in case
for t in range(T):
    yield x
    x = self.A_hat @ x

```

As desired, if we create an instance and update a primitive like  $\alpha$ , derived objects like  $A$  will also change

```

In [1]: run lake_model.py

In [2]: lm = LakeModel()

In [3]: lm.alpha
Out[3]: 0.013

In [4]: lm.A
Out[4]:
array([[ 0.97888686,  0.28067374],
       [ 0.02529314,  0.72350626]])

In [5]: lm.alpha = 2

In [6]: lm.A
Out[6]:
array([[-0.99178   ,  0.28067374],
       [ 1.99596   ,  0.72350626]])

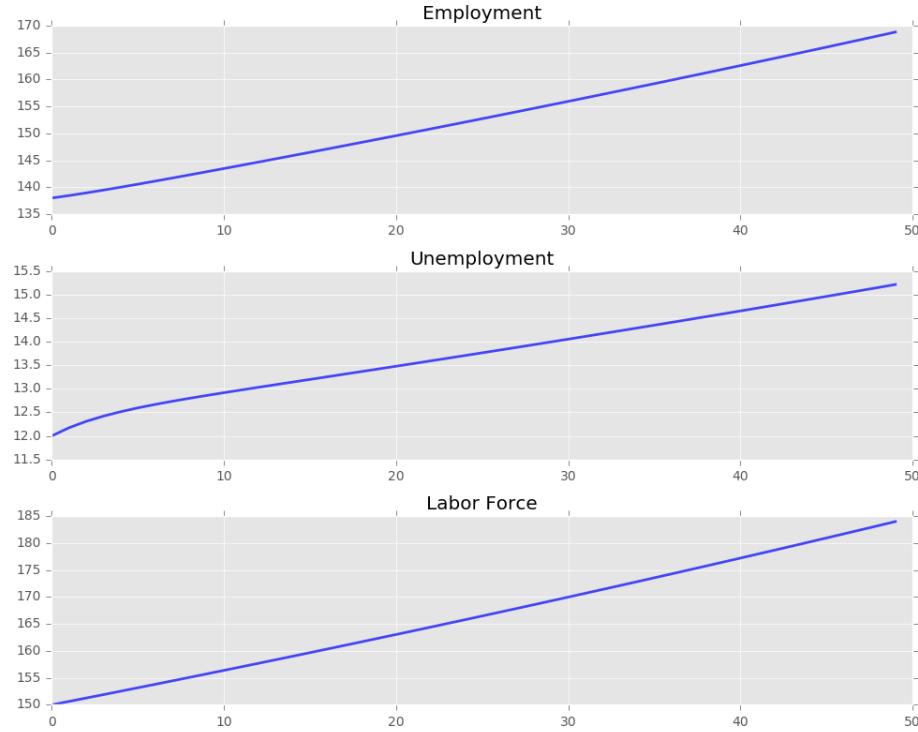
```

**Aggregate Dynamics** Let's run a simulation under the default parameters (see the class definition) starting from  $X_0 = (138, 12)$

Here's the output from `lake_stock_dynamics.py` from the [applications repository](#)

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  $x_t \rightarrow \bar{x}$  as  $t \rightarrow \infty$  provided that the remaining eigenvalue of  $\hat{A}$  has modulus less than 1

This is the case for our default parameters:

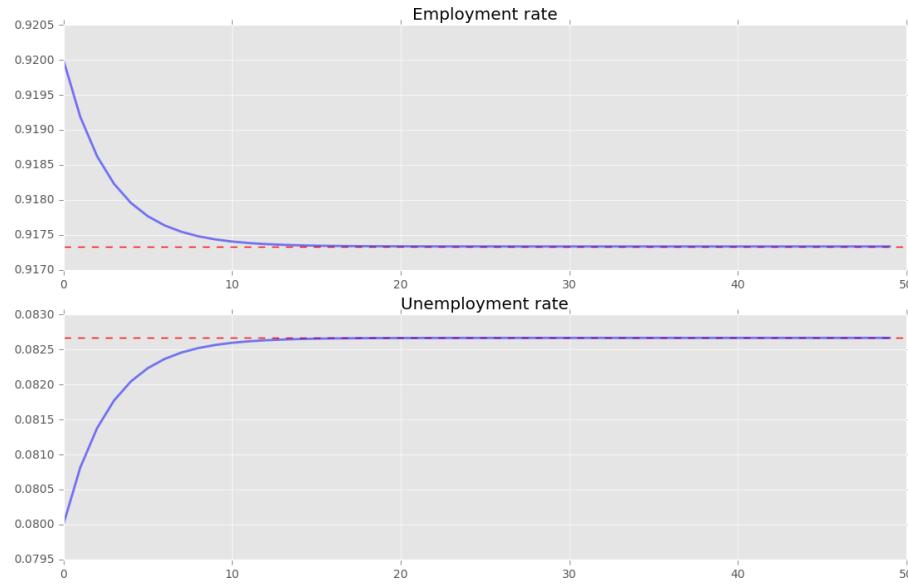
```
In [29]: run lake_model.py
In [30]: lm = LakeModel()
In [31]: e, f = np.linalg.eigvals(lm.A_hat)
In [32]: abs(e), abs(f)
Out[32]: (1.0, 0.69530673783584618)
```

The figure below illustrates the convergence of the unemployment and employment rate to steady state levels (dashed red line)

### 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  $\psi_t$  denote the *marginal distribution* over employment / unemployment states for the worker at time  $t$

As usual, we regard it as a row vector

We know from an earlier discussion that  $\psi_t$  follows the law of motion

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

We also know from the lecture on finite Markov chains that if  $\alpha \in (0, 1)$  and  $\lambda \in (0, 1)$ , then  $P$  is stationary and ergodic, with a unique stationary distribution  $\psi^*$

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

$$\lim_{T \rightarrow \infty} \bar{s}_{u,T} = \psi^*[0] \quad \text{and} \quad \lim_{T \rightarrow \infty} \bar{s}_{e,T} = \psi^*[1]$$

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

The figure below plots the path of the sample averages over 5,000 periods

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

The code that generates these plots can be found in file `lake_agent_dynamics.py` from the applications repository

### 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

- $\alpha$  (the separation rate)
- $\beta$  (the discount factor)
- $\gamma$  (the offer arrival rate)
- $c$  (unemployment compensation)

**Linking the McCall Search Model to the Lake Model** Suppose that all workers inside a Lake Model behave according to the McCall search model

The exogenous probability of leaving employment remains  $\alpha$

But their optimal decision rules determine the probability  $\lambda$  of leaving unemployment

This is now

$$\lambda = \gamma \mathbb{P}\{w_t \geq \bar{w}\} = \gamma \sum_{w' \geq \bar{w}} p(w') \quad (2.62)$$

**Fiscal Policy** We can use the McCall search version of the Lake Model to find an optimal level of unemployment insurance

We assume that the government sets unemployment compensation  $c$

The government imposes a lump sum tax  $\tau$  sufficient to finance total unemployment payments  
To attain a balanced budget at a steady state, taxes, the steady state unemployment rate  $u$ , and the  
unemployment compensation rate must satisfy

$$\tau = uc$$

The lump sum tax applies to everyone, including unemployed workers

Thus, the post-tax income of an employed worker with wage  $w$  is  $w - \tau$

The post-tax income of an unemployed worker is  $c - \tau$ .

For each specification  $(c, \tau)$  of government policy, we can solve for the worker's optimal reservation wage

This determines  $\lambda$  via (2.62) 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

$$\tau = u(c, \tau)c$$

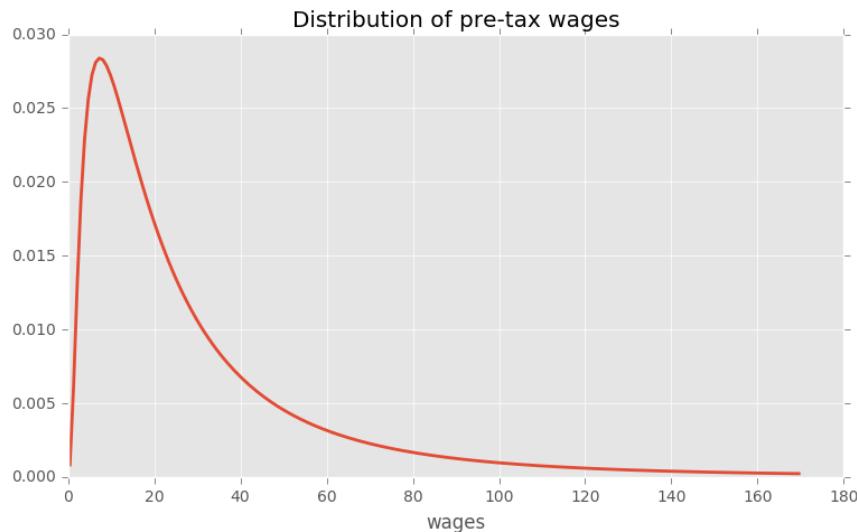
To evaluate alternative government tax-unemployment compensation pairs, we require a welfare  
criterion

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  $\alpha$ , the hazard rate of leaving employment, to

- $\alpha = 0.013$

**Fiscal Policy Code** To implement the model we use the code in `lake_fiscal_policy.py` from the applications repository

Let's have a look at this code

```
"""
Compute and plot welfare, employment, unemployment, and tax revenue as a
function of the unemployment compensation rate in the lake model.

"""

import numpy as np
import matplotlib.pyplot as plt
from lake_model import LakeModel
from scipy.stats import norm
from scipy.optimize import brentq
from numba import jit

# Make sure you have local copies of these files in your pwd
from mcall_bellman_iteration import McCallModel
from compute_reservation_wage import compute_reservation_wage

# Use ggplot style
import matplotlib
matplotlib.style.use('ggplot')

# 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()

```

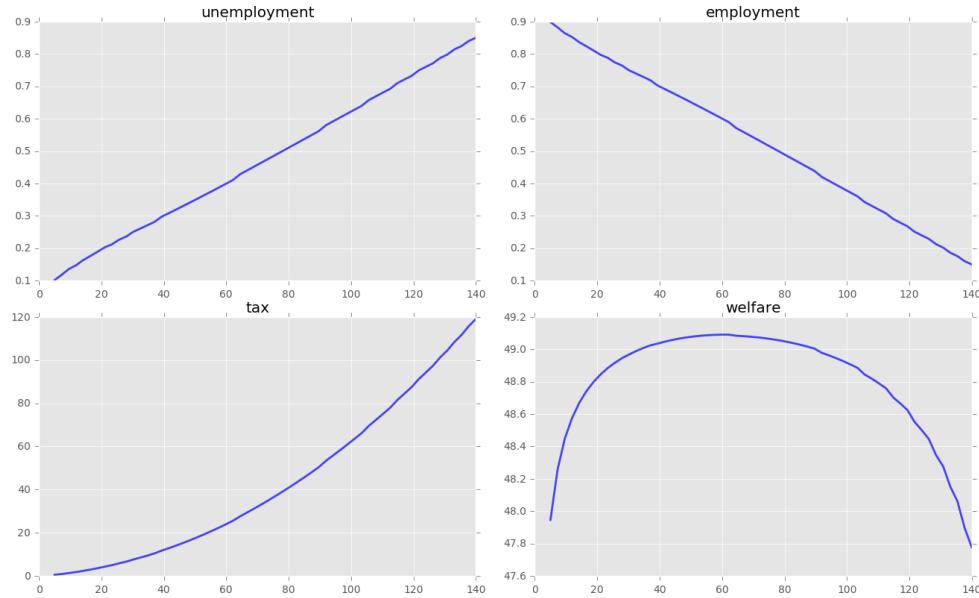
One thing to note is that the import statements assume you have copies of [the code](#) we used to solve the [McCall search model](#) in your present working directory

(We invested effort in optimizing the McCall model iteration scheme via Numba so it would be useful in more intensive computations such as this one)

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  $\alpha$  and  $\lambda$  follow [DFH06])

Suppose that in response to new legislation the hiring rate reduces to  $\lambda = 0.2$

Plot the transition dynamics of the unemployment and employment stocks for 50 periods

Plot the transition dynamics for the rates

How long does the economy take to converge to its new steady state?

What is the new steady state level of employment?

**Exercise 2** Consider an economy with initial stock of workers  $N_0 = 100$  at the steady state level of employment in the baseline parameterization

Suppose that for 20 periods the birth rate was temporarily high ( $b = 0.0025$ ) and then returned to its original level

Plot the transition dynamics of the unemployment and employment stocks for 50 periods

Plot the transition dynamics for the rates

How long does the economy take to return to its original steady state?

## Solutions

[Solution notebook](#)

# 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 last reference gives a particularly clear and 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-longitude coordinates on a map

At the present moment in time, the precise location  $x$  is unknown, but we do have some beliefs about  $x$

One way to summarize our knowledge is a point prediction  $\hat{x}$

- But what if the President wants to know the probability that the missile is currently over the Sea of Japan?
- 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, we will always assume that our prior is Gaussian. In particular, we take

$$p = N(\hat{x}, \Sigma) \quad (2.63)$$

where  $\hat{x}$  is the mean of the distribution and  $\Sigma$  is a  $2 \times 2$  covariance matrix. In our simulations, we will suppose that

$$\hat{x} = \begin{pmatrix} 0.2 \\ -0.2 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} 0.4 & 0.3 \\ 0.3 & 0.45 \end{pmatrix} \quad (2.64)$$

This density  $p(x)$  is shown below as a contour map, with the center of the red ellipse being equal to  $\hat{x}$

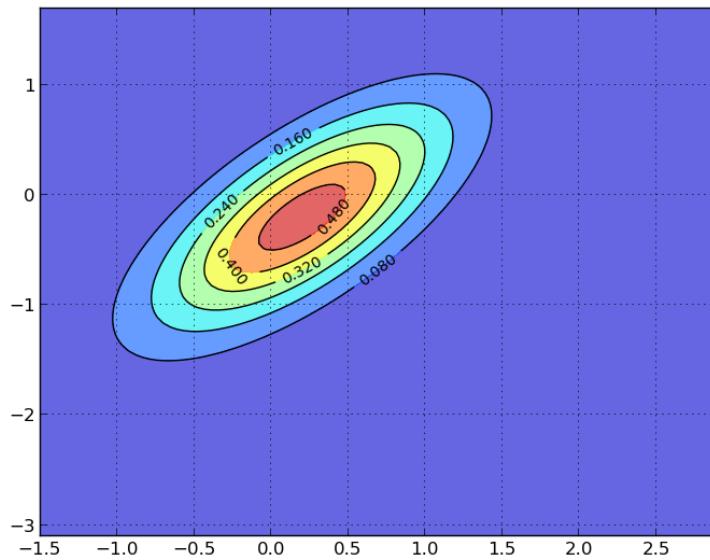


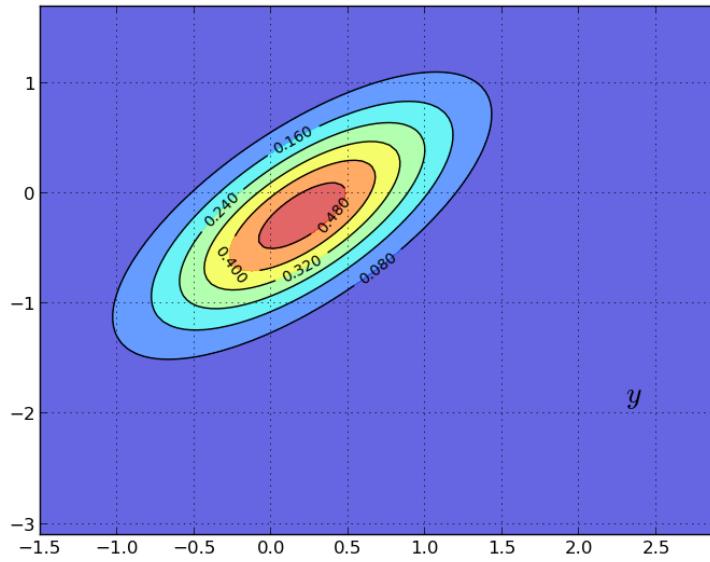
Fig. 2.1: Prior density (Click this or any other figure to enlarge.)

**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

$$y = Gx + v, \quad \text{where } v \sim N(0, R) \quad (2.65)$$

Here  $G$  and  $R$  are  $2 \times 2$  matrices with  $R$  positive definite. Both are assumed known, and the noise term  $v$  is assumed to be independent of  $x$

How then should we combine our prior  $p(x) = N(\hat{x}, \Sigma)$  and this new information  $y$  to improve our understanding of the location of the missile?

As you may have guessed, the answer is to use Bayes' theorem, which tells us we should 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 (2.65), 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 <sup>1</sup> to be

$$p(x | y) = N(\hat{x}^F, \Sigma^F)$$

---

<sup>1</sup> 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](#).

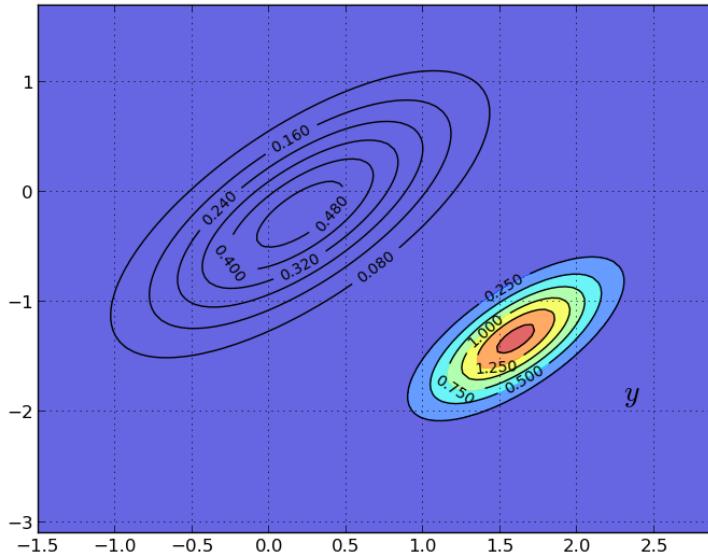
where

$$\hat{x}^F := \hat{x} + \Sigma G' (G \Sigma G' + R)^{-1} (y - G \hat{x}) \quad \text{and} \quad \Sigma^F := \Sigma - \Sigma G' (G \Sigma G' + R)^{-1} G \Sigma \quad (2.66)$$

Here  $\Sigma G' (G \Sigma G' + R)^{-1}$  is the matrix of population regression coefficients of the hidden object  $x - \hat{x}$  on the surprise  $y - G \hat{x}$

This new density  $p(x | y) = N(\hat{x}^F, \Sigma^F)$  is shown in the next figure via contour lines and the color map

The original density is left in as contour lines for comparison



Our new density twists the prior  $p(x)$  in a direction determined by the new information  $y - G \hat{x}$

In generating the figure, we set  $G$  to the identity matrix and  $R = 0.5\Sigma$  for  $\Sigma$  defined in (2.64)

(The code for generating this and the preceding figures can be found in the file `gaussian_contours.py` from the `QuantEcon.applications` package

### 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

- $p(x | y) = N(\hat{x}^F, \Sigma^F)$  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,

$$x_{t+1} = Ax_t + w_{t+1}, \quad \text{where } w_t \sim N(0, Q) \quad (2.67)$$

Our aim is to combine this law of motion and our current distribution  $p(x|y) = N(\hat{x}^F, \Sigma^F)$  to come up with a new *predictive* distribution for the location one unit of time hence

In view of (2.67), 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 (2.66) 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

$$\text{Var}[Ax^F + w] = A \text{Var}[x^F]A' + Q = A\Sigma^F A' + Q = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q$$

The matrix  $A\Sigma G'(G\Sigma G' + R)^{-1}$  is often written as  $K_\Sigma$  and called the *Kalman gain*

- the subscript  $\Sigma$  has been added to remind us that  $K_\Sigma$  depends on  $\Sigma$ , but not  $y$  or  $\hat{x}$

Using this notation, we can summarize our results as follows: Our updated prediction is the density  $N(\hat{x}_{new}, \Sigma_{new})$  where

$$\begin{aligned} \hat{x}_{new} &:= A\hat{x} + K_\Sigma(y - G\hat{x}) \\ \Sigma_{new} &:= A\Sigma A' - K_\Sigma G\Sigma A' + Q \end{aligned} \quad (2.68)$$

- 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

$$A = \begin{pmatrix} 1.2 & 0.0 \\ 0.0 & -0.2 \end{pmatrix}, \quad Q = 0.3 * \Sigma$$

**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

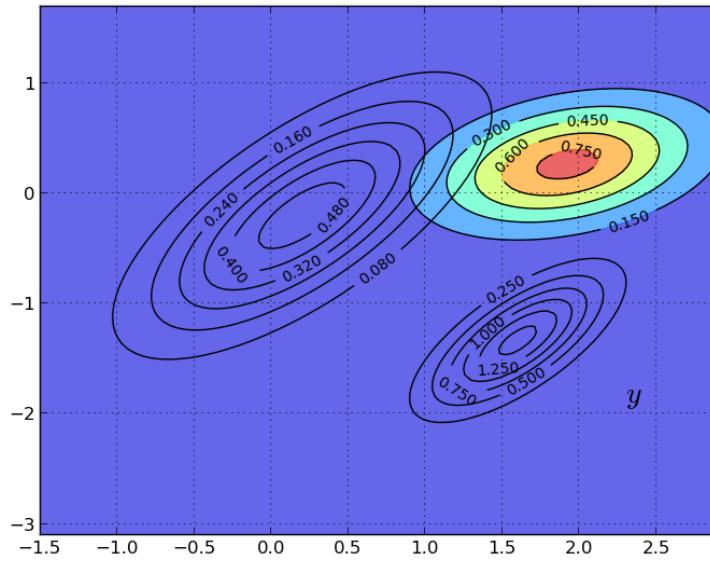
We then used the current measurement  $y$  to update to  $p(x|y)$

Finally, we used the law of motion (2.67) for  $\{x_t\}$  to update to  $p_{new}(x)$

If we now step into the next period, we are ready to go round again, taking  $p_{new}(x)$  as the current prior

Swapping notation  $p_t(x)$  for  $p(x)$  and  $p_{t+1}(x)$  for  $p_{new}(x)$ , the full recursive procedure is:

1. Start the current period with prior  $p_t(x) = N(\hat{x}_t, \Sigma_t)$
2. Observe current measurement  $y_t$



3. Compute the filtering distribution  $p_t(x | y) = N(\hat{x}_t^F, \Sigma_t^F)$  from  $p_t(x)$  and  $y_t$ , applying Bayes rule and the conditional distribution (2.65)
4. Compute the predictive distribution  $p_{t+1}(x) = N(\hat{x}_{t+1}, \Sigma_{t+1})$  from the filtering distribution and (2.67)
5. Increment  $t$  by one and go to step 1

Repeating (2.68), the dynamics for  $\hat{x}_t$  and  $\Sigma_t$  are as follows

$$\begin{aligned}\hat{x}_{t+1} &= A\hat{x}_t + K_{\Sigma_t}(y_t - G\hat{x}_t) \\ \Sigma_{t+1} &= A\Sigma_t A' - K_{\Sigma_t}G\Sigma_t A' + Q\end{aligned}\tag{2.69}$$

These are the standard dynamic equations for the Kalman filter. See, for example, [LS12], page 58.

### Convergence

The matrix  $\Sigma_t$  is a measure of the uncertainty of our prediction  $\hat{x}_t$  of  $x_t$

Apart from special cases, this uncertainty will never be fully resolved, regardless of how much time elapses

One reason is that our prediction  $\hat{x}_t$  is made based on information available at  $t - 1$ , not  $t$

Even if we know the precise value of  $x_{t-1}$  (which we don't), the transition equation (2.67) implies that  $x_t = Ax_{t-1} + w_t$

Since the shock  $w_t$  is not observable at  $t - 1$ , any time  $t - 1$  prediction of  $x_t$  will incur some error (unless  $w_t$  is degenerate)

However, it is certainly possible that  $\Sigma_t$  converges to a constant matrix as  $t \rightarrow \infty$

To study this topic, let's expand the second equation in (2.69):

$$\Sigma_{t+1} = A\Sigma_t A' - A\Sigma_t G'(G\Sigma_t G' + R)^{-1}G\Sigma_t A' + Q \quad (2.70)$$

This is a nonlinear difference equation in  $\Sigma_t$

A fixed point of (2.70) is a constant matrix  $\Sigma$  such that

$$\Sigma = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q \quad (2.71)$$

Equation (2.70) is known as a discrete time Riccati difference equation

Equation (2.71) 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

One sufficient (but not necessary) condition is that all the eigenvalues  $\lambda_i$  of  $A$  satisfy  $|\lambda_i| < 1$  (cf. e.g., [AM05], p. 77)

(This strong condition assures that the unconditional distribution of  $x_t$  converges as  $t \rightarrow +\infty$ )

In this case, for any initial choice of  $\Sigma_0$  that is both nonnegative and symmetric, the sequence  $\{\Sigma_t\}$  in (2.70) converges to a nonnegative symmetric matrix  $\Sigma$  that solves (2.71)

### Implementation

The class `Kalman` from the `QuantEcon.py` package implements the Kalman filter

- Instance data consists of:
  - the moments  $(\hat{x}_t, \Sigma_t)$  of the current prior
  - An instance of the `LinearStateSpace` class from `QuantEcon.py`

The latter represents a linear state space model of the form

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + Hv_t \end{aligned}$$

where the shocks  $w_t$  and  $v_t$  are iid standard normals

To connect this with the notation of this lecture we set

$$Q := CC' \quad \text{and} \quad R := HH'$$

- The main methods 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 (2.71) and the corresponding (stationary) Kalman gain

You can view the program [on GitHub](#) but we repeat it here for convenience

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

Implements the Kalman filter for a linear Gaussian state space model.

"""

from textwrap import dedent
import numpy as np
from numpy import dot
from scipy.linalg import inv
from quantecon.lss import LinearStateSpace
from quantecon.matrix_eqn import solve_discrete_riccati

class Kalman(object):
    """
    Implements the Kalman filter for the Gaussian state space model

    
$$\begin{aligned} x_{t+1} &= A x_t + C w_{t+1} \\ y_t &= G x_t + H v_t. \end{aligned}$$


    Here  $x_t$  is the hidden state and  $y_t$  is the measurement. The shocks  $w_t$  and  $v_t$  are iid standard normals. Below we use the notation

    
$$\begin{aligned} Q &:= CC' \\ R &:= HH' \end{aligned}$$


    Parameters
    -----
    ss : instance of LinearStateSpace
        An instance of the quantecon.lss.LinearStateSpace class
    x_hat : scalar(float) or array_like(float), optional(default=None)
        An  $n \times 1$  array representing the mean  $x_{\text{hat}}$  and covariance
        matrix  $\Sigma$  of the prior/predictive density. Set to zero if
        not supplied.
    Sigma : scalar(float) or array_like(float), optional(default=None)
        An  $n \times n$  array representing the covariance matrix  $\Sigma$  of
        the prior/predictive density. Must be positive definite.
        Set to the identity if not supplied.

    Attributes
    -----
    Sigma, x_hat : as above
    Sigma_infinity : array_like or scalar(float)
        The infinite limit of  $\Sigma_t$ 
    K_infinity : array_like or scalar(float)
        The stationary Kalman gain.
```

*References*

-----

<http://quant-econ.net/py/kalman.html>

"""

```

def __init__(self, ss, x_hat=None, Sigma=None):
    self.ss = ss
    self.set_state(x_hat, Sigma)
    self.K_infinity = None
    self.Sigma_infinity = None

def set_state(self, x_hat, Sigma):
    if Sigma is None:
        Sigma = np.identity(self.ss.n)
    else:
        self.Sigma = np.atleast_2d(Sigma)
    if x_hat is None:
        x_hat = np.zeros((self.ss.n, 1))
    else:
        self.x_hat = np.atleast_2d(x_hat)
        self.x_hat.shape = self.ss.n, 1

def __repr__(self):
    return self.__str__()

def __str__(self):
    m = """\
    Kalman filter:
    - dimension of state space           : {n}
    - dimension of observation equation : {k}
    """
    return dedent(m.format(n=self.ss.n, k=self.ss.k))

def whitener_lss(self):
    r"""
    This function takes the linear state space system
    that is an input to the Kalman class and it converts
    that system to the time-invariant whitener representation
    given by

    \tilde{x}_{t+1}^* = \tilde{A} \tilde{x}_t + \tilde{C} v
    a = \tilde{G} \tilde{x}_t

    where

    \tilde{x}_t = [x_t, \hat{x}_t, v_t]
    and

    \tilde{A} = [A 0 0
                0 0 0
                0 0 0]
    """

```

```

    KG  A-KG  KH
    0    0     0]

\tilde{C} = [C  0
            0  0
            0  I]

\tilde{G} = [G -G  H]

with A, C, G, H coming from the linear state space system
that defines the Kalman instance

>Returns
-----
whitened_lss : LinearStateSpace
    This is the linear state space system that represents
    the whitened system
"""
# Check for steady state Sigma and K
if self.K_infinity is None:
    Sig, K = self.stationary_values()
    self.Sigma_infinity = Sig
    self.K_infinity = K
else:
    K = self.K_infinity

# Get the matrix sizes
n, k, m, l = self.ss.n, self.ss.k, self.ss.m, self.ss.l
A, C, G, H = self.ss.A, self.ss.C, self.ss.G, self.ss.H

Atil = np.vstack([np.hstack([A, np.zeros((n, n)), np.zeros((n, 1))]),
                  np.hstack([dot(K, G), A-dot(K, G), dot(K, H)]),
                  np.zeros((l, 2*n + 1))])

Ctil = np.vstack([np.hstack([C, np.zeros((n, 1))]),
                  np.zeros((n, m+1)),
                  np.hstack([np.zeros((l, m)), np.eye(l)])])

Gtil = np.hstack([G, -G, H])

whitened_lss = LinearStateSpace(Atil, Ctil, Gtil)
self.whitened_lss = whitened_lss

return whitened_lss

def prior_to_filtered(self, y):
    """
    Updates the moments (x_hat, Sigma) of the time t prior to the
    time t filtering distribution, using current measurement y_t.

    The updates are according to

```

```


$$\hat{x}_F = \hat{x} + \Sigma G' (G \Sigma G' + R)^{-1}$$


$$(y - G \hat{x})$$


$$\Sigma^F = \Sigma - \Sigma G' (G \Sigma G' + R)^{-1} G$$


$$\Sigma$$


Parameters
-----
y : scalar or array_like(float)
    The current measurement

"""
# === simplify notation === #
G, H = self.ss.G, self.ss.H
R = np.dot(H, H.T)

# === and then update === #
y = np.atleast_2d(y)
y.shape = self.ss.k, 1
E = dot(self.Sigma, G.T)
F = dot(dot(G, self.Sigma), G.T) + R
M = dot(E, inv(F))
self.x_hat = self.x_hat + dot(M, (y - dot(G, self.x_hat)))
self.Sigma = self.Sigma - dot(M, dot(G, self.Sigma))

def filtered_to_forecast(self):
    """
    Updates the moments of the time t filtering distribution to the
    moments of the predictive distribution, which becomes the time
    t+1 prior

    """
    # === simplify notation === #
    A, C = self.ss.A, self.ss.C
    Q = np.dot(C, C.T)

    # === and then update === #
    self.x_hat = dot(A, self.x_hat)
    self.Sigma = dot(A, dot(self.Sigma, A.T)) + Q

def update(self, y):
    """
    Updates x_hat and Sigma given k x 1 ndarray y. The full
    update, from one period to the next

    Parameters
    -----
    y : np.ndarray
        A k x 1 ndarray y representing the current measurement

    """
    self.prior_to_filtered(y)
    self.filtered_to_forecast()

```

```

def stationary_values(self):
    """
    Computes the limit of Sigma_t as t goes to infinity by
    solving the associated Riccati equation. Computation is via the
    doubling algorithm (see the documentation in
    `matrix_eqn.solve_discrete_riccati`).

    Returns
    ------
    Sigma_infinity : array_like or scalar(float)
        The infinite limit of Sigma_t
    K_infinity : array_like or scalar(float)
        The stationary Kalman gain.

    """
    # === simplify notation === #
    A, C, G, H = self.ss.A, self.ss.C, self.ss.G, self.ss.H
    Q, R = np.dot(C, C.T), np.dot(H, H.T)

    # === solve Riccati equation, obtain Kalman gain === #
    Sigma_infinity = solve_discrete_riccati(A.T, G.T, Q, R)
    temp1 = dot(dot(A, Sigma_infinity), G.T)
    temp2 = inv(dot(G, dot(Sigma_infinity, G.T)) + R)
    K_infinity = dot(temp1, temp2)

    # == record as attributes and return ==
    self.Sigma_infinity, self.K_infinity = Sigma_infinity, K_infinity
    return Sigma_infinity, K_infinity

def stationary_coefficients(self, j, coeff_type='ma'):
    """
    Hold representation moving average or VAR coefficients for the
    steady state Kalman filter.

    Parameters
    -----
    j : int
        The lag length
    coeff_type : string, either 'ma' or 'var' (default='ma')
        The type of coefficient sequence to compute. Either 'ma' for
        moving average or 'var' for VAR.
    """
    # == simplify notation ==
    A, G = self.ss.A, self.ss.G
    K_infinity = self.K_infinity
    # == make sure that K_infinity has actually been computed ==
    if K_infinity is None:
        S, K_infinity = self.stationary_values()
    # == compute and return coefficients ==
    coeffs = []
    i = 1
    if coeff_type == 'ma':
        coeffs.append(np.identity(self.ss.k))

```

```

P_mat = A
P = np.identity(self.ss.n) # Create a copy
elif coeff_type == 'var':
    coeffs.append(dot(G, K_infinity))
    P_mat = A - dot(K_infinity, G)
    P = np.copy(P_mat) # Create a copy
else:
    raise ValueError("Unknown coefficient type")
while i <= j:
    coeffs.append(dot(dot(G, P), K_infinity))
    P = dot(P, P_mat)
    i += 1
return coeffs

def stationary_innovation_covar(self):
    # == simplify notation == #
    H, G = self.ss.H, self.ss.G
    R = np.dot(H, H.T)
    Sigma_infinity = self.Sigma_infinity

    # == make sure that Sigma_infinity has been computed == #
    if Sigma_infinity is None:
        Sigma_infinity, K = self.stationary_values()
    return dot(G, dot(Sigma_infinity, G.T)) + R

```

## 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 (2.67) 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

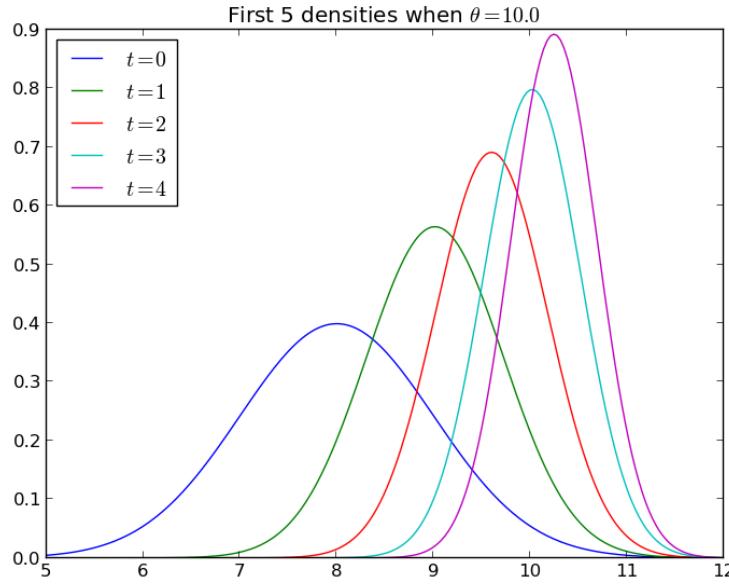
The task of this exercise to simulate the model and, using the code from `kalman.py`, plot the first five predictive densities  $p_t(x) = N(\hat{x}_t, \Sigma_t)$

As shown in [LS12], sections 2.9.1–2.9.2, these distributions asymptotically put all mass on the unknown value  $\theta$

In the simulation, take  $\theta = 10$ ,  $\hat{x}_0 = 8$  and  $\Sigma_0 = 1$

Your figure should – modulo randomness – look something like this

**Exercise 2** The preceding figure gives some support to the idea that probability mass converges to  $\theta$



To get a better idea, choose a small  $\epsilon > 0$  and calculate

$$z_t := 1 - \int_{\theta-\epsilon}^{\theta+\epsilon} p_t(x) dx$$

for  $t = 0, 1, 2, \dots, T$

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  $Ax_{t-1}$

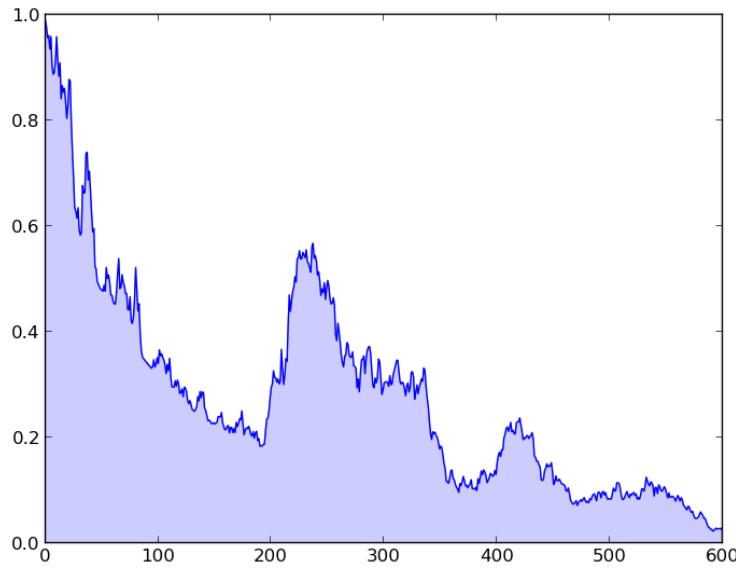
The conditional expectation is known to be the optimal prediction method in terms of minimizing mean squared error

(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  $t = 1, \dots, 50$



For the parameters, set  $G = I$ ,  $R = 0.5I$  and  $Q = 0.3I$ , where  $I$  is the  $2 \times 2$  identity

Set

$$A = \begin{pmatrix} 0.5 & 0.4 \\ 0.6 & 0.3 \end{pmatrix}$$

To initialize the prior density, set

$$\Sigma_0 = \begin{pmatrix} 0.9 & 0.3 \\ 0.3 & 0.9 \end{pmatrix}$$

and  $\hat{x}_0 = (8, 8)$

Finally, set  $x_0 = (0, 0)$

You should end up with a figure similar to the following (modulo randomness)

Observe how, after an initial learning period, the Kalman filter performs quite well, even relative to the competitor who predicts optimally with knowledge of the latent state

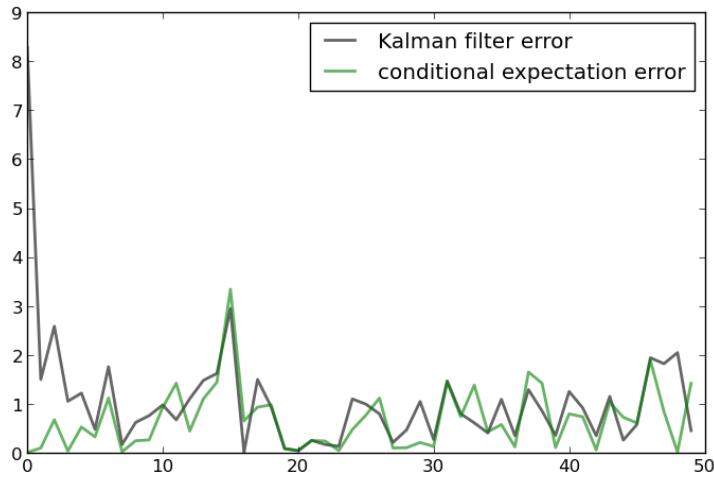
**Exercise 4** Try varying the coefficient 0.3 in  $Q = 0.3I$  up and down

Observe how the diagonal values in the stationary solution  $\Sigma$  (see (2.71)) 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

[Solution notebook](#)



## 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 entrepreneurs – have beliefs about the fundamentals expressed as probability distributions
- Greater uncertainty means greater dispersions of these distributions
- Entrepreneurs are risk averse and hence less inclined to be active when uncertainty is high
- The output of active entrepreneurs is observable, supplying a noisy signal that helps everyone inside the model infer fundamentals
- Entrepreneurs update their beliefs about fundamentals using Bayes' Law, implemented via Kalman filtering

Uncertainty traps emerge because:

- High uncertainty discourages entrepreneurs from becoming active
- A low level of participation – i.e., a smaller number of active entrepreneurs – diminishes the flow of information about fundamentals

- Less information translates to higher uncertainty, further discouraging entrepreneurs from choosing to be active, and so on

Uncertainty traps stem from a positive externality: high aggregate economic activity levels generates valuable information

### 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  $\theta_t$  is not observable at any time

**Output** There is a total  $\bar{M}$  of risk averse entrepreneurs

Output of the  $m$ -th entrepreneur, conditional on being active in the market at time  $t$ , is equal to

$$x_m = \theta + \epsilon_m \quad \text{where} \quad \epsilon_m \sim N(0, \gamma_x^{-1}) \quad (2.72)$$

Here the time subscript has been dropped to simplify notation

The inverse of the shock variance,  $\gamma_x$ , 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  $\theta_0$

Signals are publicly observable and hence all agents have identical beliefs always

Dropping time subscripts, beliefs for current  $\theta$  are represented by the normal distribution  $N(\mu, \gamma^{-1})$

Here  $\gamma$  is the precision of beliefs; its inverse is the degree of uncertainty

These parameters are updated by Kalman filtering

Let

- $\mathbb{M} \subset \{1, \dots, \bar{M}\}$  denote the set of currently active firms
- $M := |\mathbb{M}|$  denote the number of currently active firms

- $X$  be the average output  $\frac{1}{M} \sum_{m \in M} x_m$  of the active firms

With this notation and primes for next period values, we can write the updating of the mean and precision via

$$\mu' = \rho \frac{\gamma \mu + M \gamma_x X}{\gamma + M \gamma_x} \quad (2.73)$$

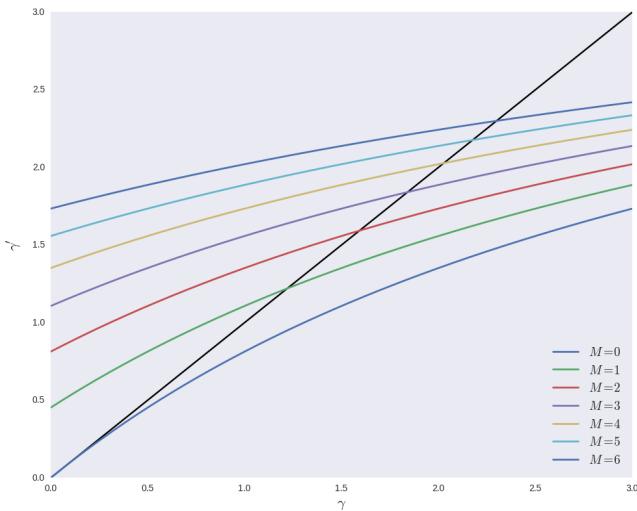
$$\gamma' = \left( \frac{\rho^2}{\gamma + M \gamma_x} + \sigma_\theta^2 \right)^{-1} \quad (2.74)$$

These are standard Kalman filtering results applied to the current setting

Exercise 1 provides more details on how (2.73) and (2.74) are derived, and then asks you to fill in remaining steps

The next figure plots the law of motion for the precision in (2.74) as a 45 degree diagram, with one curve for each  $M \in \{0, \dots, 6\}$

The other parameter values are  $\rho = 0.99, \gamma_x = 0.5, \sigma_\theta = 0.5$



Points where the curves hit the 45 degree lines are long run steady states for precision for different values of  $M$

Thus, if one of these values for  $M$  remains fixed, a corresponding steady state is the equilibrium level of precision

- high values of  $M$  correspond to greater information about the fundamental, and hence more precision in steady state
- low values of  $M$  correspond to less information and more uncertainty in steady state

In practice, as we'll see, the number of active firms fluctuates stochastically

**Participation** Omitting time subscripts once more, entrepreneurs enter the market in the current period if

$$\mathbb{E}[u(x_m - F_m)] > c \quad (2.75)$$

Here

- the mathematical expectation of  $x_m$  is based on (2.72) and beliefs  $N(\mu, \gamma^{-1})$  for  $\theta$
- $F_m$  is a stochastic but previsible fixed cost, independent across time and firms
- $c$  is a constant reflecting opportunity costs

The statement that  $F_m$  is previsible means that it is realized at the start of the period and treated as a constant in (2.75)

The utility function has the constant absolute risk aversion form

$$u(x) = \frac{1}{a} (1 - \exp(-ax)) \quad (2.76)$$

where  $a$  is a positive parameter

Combining (2.75) and (2.76), entrepreneur  $m$  participates in the market (or is said to be active) when

$$\frac{1}{a} \{1 - \mathbb{E}[\exp(-a(\theta + \epsilon_m - F_m))]\} > c$$

Using standard formulas for expectations of lognormal random variables, this is equivalent to the condition

$$\psi(\mu, \gamma, F_m) := \frac{1}{a} \left( 1 - \exp \left( -a\mu + aF_m + \frac{a^2 \left( \frac{1}{\gamma} + \frac{1}{\gamma_x} \right)}{2} \right) \right) - c > 0 \quad (2.77)$$

## Implementation

We want to simulate this economy

As a first step, let's put together a class that bundles

- the parameters, the current value of  $\theta$  and the current values of the two belief parameters  $\mu$  and  $\gamma$
- methods to update  $\theta$ ,  $\mu$  and  $\gamma$ , as well as to determine the number of active firms and their outputs

The updating methods follow the laws of motion for  $\theta$ ,  $\mu$  and  $\gamma$  given above

The method to evaluate the number of active firms generates  $F_1, \dots, F_M$  and tests condition (2.77) 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 also 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  $\mu$ , which the agents use to track  $\theta$



We see that  $\mu$  tracks  $\theta$  well when there are sufficient firms in the market

However, there are times when  $\mu$  tracks  $\theta$  poorly due to insufficient information

These are episodes where the uncertainty traps take hold

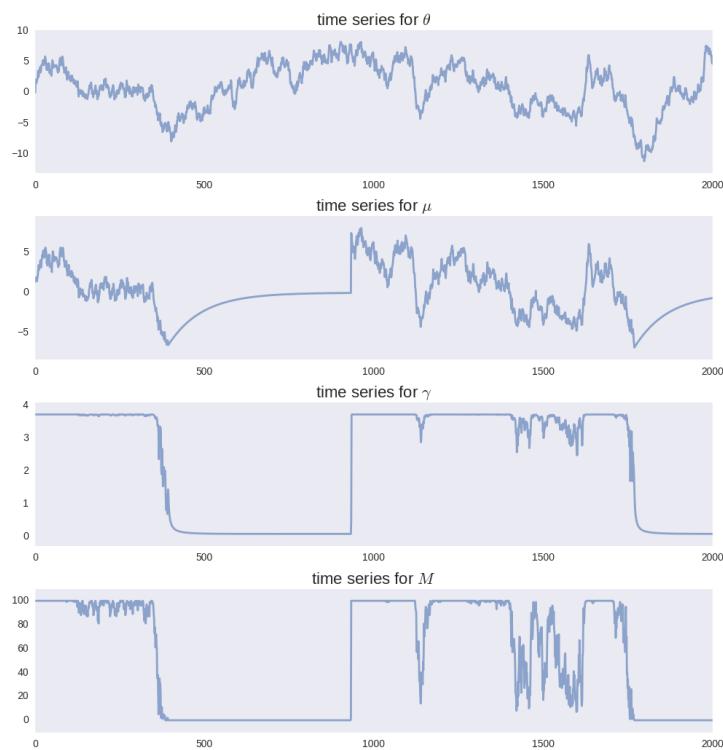
During these episodes

- precision is low and uncertainty is high
- few firms are in the market

To get a clearer idea of the dynamics, let's look at all the main time series at once, for a given set of shocks

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 (2.73) and (2.74) based on the following standard result (see, e.g., p. 24 of [YS05])

**Fact** Let  $\mathbf{x} = (x_1, \dots, x_M)$  be a vector of IID draws from common distribution  $N(\theta, 1/\gamma_x)$  and let  $\bar{x}$  be the sample mean. If  $\gamma_x$  is known and the prior for  $\theta$  is  $N(\mu, 1/\gamma)$ , then the posterior distribution of  $\theta$  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

[Solution notebook](#)

## A Simple Optimal Growth Model

### Contents

- *A Simple Optimal Growth Model*
  - [Overview](#)
  - [The Model](#)
  - [Dynamic Programming](#)
  - [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 model is intentionally simplistic — for now we favor ease of exposition over realism

The technique we use to solve the model is dynamic programming, which is

- pervasive in economics, finance and many other fields
- general and powerful, yielding both intuition and practical computational methods

Our treatment of dynamic programming follows on from earlier, less technical treatments in our lectures on [shortest paths](#) and [job search](#)

### The Model

Consider an agent who owns at time  $t$  capital stock  $k_t \in \mathbb{R}_+ := [0, \infty)$  and produces output

$$y_t := f(k_t) \in \mathbb{R}_+$$

This output can either be consumed or saved as capital for next period

For simplicity we assume that depreciation is total, so that next period capital is just output minus consumption:

$$k_{t+1} = y_t - c_t \quad (2.78)$$

Taking  $k_0$  as given, we suppose that the agent wishes to maximize

$$\sum_{t=0}^{\infty} \beta^t u(c_t) \quad (2.79)$$

where  $u$  is a given utility function and  $\beta \in (0, 1)$  is a discount factor

More precisely, the agent wishes to select a path  $c_0, c_1, c_2, \dots$  for consumption that is

1. nonnegative
2. feasible, in the sense that the capital path  $\{k_t\}$  determined by  $\{c_t\}$ ,  $k_0$  and (2.78) is always nonnegative
3. optimal, in the sense that it maximizes (2.79) relative to all other feasible consumption sequences

In the present context

- $k$  is called the *state* variable — it summarizes the “state of the world” at the start of each period
- $c$  is called the *control* variable — a value chosen by the agent each period after observing the state

A well-known result from dynamic programming theory (cf., e.g., [SLP89], section 4.1) states that, for kind of this problem, any optimal consumption sequence  $\{c_t\}$  is *Markov*

That is, there exists a function  $\sigma$  such that

$$c_t = \sigma(k_t) \quad \text{for all } t$$

In other words, the current control is a fixed (i.e., time homogeneous) function of the current state

**The Policy Function Approach** As it turns out, we are better off seeking the function  $\sigma$  directly, rather than the optimal consumption sequence

The main reason is that the functional approach — seeking the optimal policy — translates directly over to the stochastic case, whereas the sequential approach does not

For this model, we will say that function  $\sigma$  mapping  $\mathbb{R}_+$  into  $\mathbb{R}_+$  is a *feasible consumption policy* if it satisfies

$$\sigma(k) \leq f(k) \quad \text{for all } k \in \mathbb{R}_+ \quad (2.80)$$

The set of all such policies will be denoted by  $\Sigma$

Using this notation, the agent's decision problem can be rewritten as

$$\max_{\sigma \in \Sigma} \left\{ \sum_{t=0}^{\infty} \beta^t u(\sigma(k_t)) \right\} \quad (2.81)$$

where the sequence  $\{k_t\}$  in (2.81) is given by

$$k_{t+1} = f(k_t) - \sigma(k_t), \quad k_0 \text{ given} \quad (2.82)$$

In the next section we discuss how to solve this problem for the maximizing  $\sigma$

### Dynamic Programming

We will solve for the optimal policy using *dynamic programming*

The first step is to define the *policy value function*  $v_\sigma$  associated with a given policy  $\sigma$ , which is

$$v_\sigma(k_0) := \sum_{t=0}^{\infty} \beta^t u(\sigma(k_t)) \quad (2.83)$$

when  $\{k_t\}$  is given by (2.82)

Evidently  $v_\sigma(k_0)$  is the total present value of discounted utility associated with following policy  $\sigma$  forever, given initial capital  $k_0$

The *value function* for this optimization problem is then defined as

$$v^*(k_0) := \sup_{\sigma \in \Sigma} v_\sigma(k_0) \quad (2.84)$$

The value function gives the maximal value that can be obtained from state  $k_0$ , after considering all feasible policies

A policy  $\sigma \in \Sigma$  is called *optimal* if it attains the supremum in (2.84) for all  $k_0 \in \mathbb{R}_+$

The *Bellman equation* for this problem takes the form

$$v^*(k) = \max_{0 \leq c \leq f(k)} \{u(c) + \beta v^*(f(k) - c)\} \quad \text{for all } k \in \mathbb{R}_+ \quad (2.85)$$

It states that maximal value from a given state can be obtained by trading off

- current reward from a given action (in this case utility from current consumption) vs

- the discounted future value of the state resulting from that action

(If the intuition behind the Bellman equation is not clear to you, try working through *this lecture*)

As a matter of notation, given a continuous function  $w$  on  $\mathbb{R}_+$ , we say that policy  $\sigma \in \Sigma$  is  $w$ -greedy if  $\sigma(k)$  is a solution to

$$\max_{0 \leq c \leq f(k)} \{u(c) + \beta w(f(k) - c)\} \quad (2.86)$$

for every  $k \in \mathbb{R}_+$

**Theoretical Results** As with most optimization problems, conditions for existence of a solution typically require some form of continuity and compactness

In addition, some restrictions are needed to ensure that the sum of discounted utility is always finite

For example, if we are prepared to assume that  $f$  and  $u$  are continuous and  $u$  is bounded, then

1. The value function  $v^*$  is finite, bounded, continuous and satisfies the Bellman equation
2. At least one optimal policy exists
3. A policy is optimal if and only if it is  $v^*$ -greedy

(For a proof see, for example, proposition 10.1.13 of [EDTC](#))

In view of these results, to find an optimal policy, one option — perhaps the most common — is to

1. compute  $v^*$
2. solve for a  $v^*$ -greedy policy

The advantage is that, once we get to the second step, we are solving a one-dimensional optimization problem — the problem on the right-hand side of (2.85)

This is much easier than an infinite-dimensional optimization problem, which is what we started out with

(An infinite sequence  $\{c_t\}$  is a point in an infinite-dimensional space)

In fact step 2 is almost trivial once  $v^*$  is obtained

For this reason, most of our focus is on the first step — how to obtain the value function

**Value Function Iteration** The value function  $v^*$  can be obtained by an iterative technique:

- Start with a guess — some initial function  $w$
- successively improve it

The improvement step involves applying an operator (i.e., a map from functions to functions) called the *Bellman operator*

The Bellman operator for this problem is denoted  $T$  and sends  $w$  into  $Tw$  via

$$Tw(k) := \max_{0 \leq c \leq f(k)} \{u(c) + \beta w(f(k) - c)\} \quad (2.87)$$

Now let  $w$  be any continuous bounded function

Iteratively applying  $T$  from initial condition  $w$  produces a sequence of functions  $w, Tw, T(Tw) = T^2w, \dots$  that converges uniformly to  $v^*$

For a proof see, for example, lemma 10.1.20 of [EDTC](#)

This convergence will be prominent in our numerical experiments

**Unbounded Utility** The theoretical results stated above assume that the utility function is bounded

In practice economists often work with unbounded utility functions

For utility functions that are bounded below (but possibly unbounded above), a clean and comprehensive theory now exists

(Section 12.2 of [EDTC](#) provides one exposition)

For utility functions that are unbounded both below and above the situation is more complicated

For recent work on deterministic problems, see, for example, [\[Kam12\]](#) or [\[MdRV10\]](#)

In this lecture we will use both bounded and unbounded utility functions without dwelling on the theory

### 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 iterating with the Bellman operator

In theory, the algorithm is as follows

1. Begin with a function  $w$  — an initial condition
2. Solving (2.87), obtain the function  $Tw$
3. Unless some stopping condition is satisfied, set  $w = Tw$  and go to step 2

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 (2.87)

Even if  $w$  is a known function, unless  $Tw$  can be shown to have some special structure, the only way to store this function is to record the value  $Tw(k)$  for every  $k \in \mathbb{R}_+$

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  $\{k_1, \dots, k_I\} \subset \mathbb{R}_+$ , and reconstruct it from this information when required

More precisely, the algorithm will be

1. Begin with an array of values  $\{w_1, \dots, w_I\}$ , typically representing the values of some initial function  $w$  on the grid points  $\{k_1, \dots, k_I\}$
2. build a function  $\hat{w}$  on the state space  $\mathbb{R}_+$  by interpolating the points  $\{w_1, \dots, w_I\}$
3. By repeatedly solving (2.87), obtain and record the value  $T\hat{w}(k_i)$  on each grid point  $k_i$
4. Unless some stopping condition is satisfied, set  $\{w_1, \dots, w_I\} = \{T\hat{w}(k_1), \dots, T\hat{w}(k_I)\}$  and go to step 2

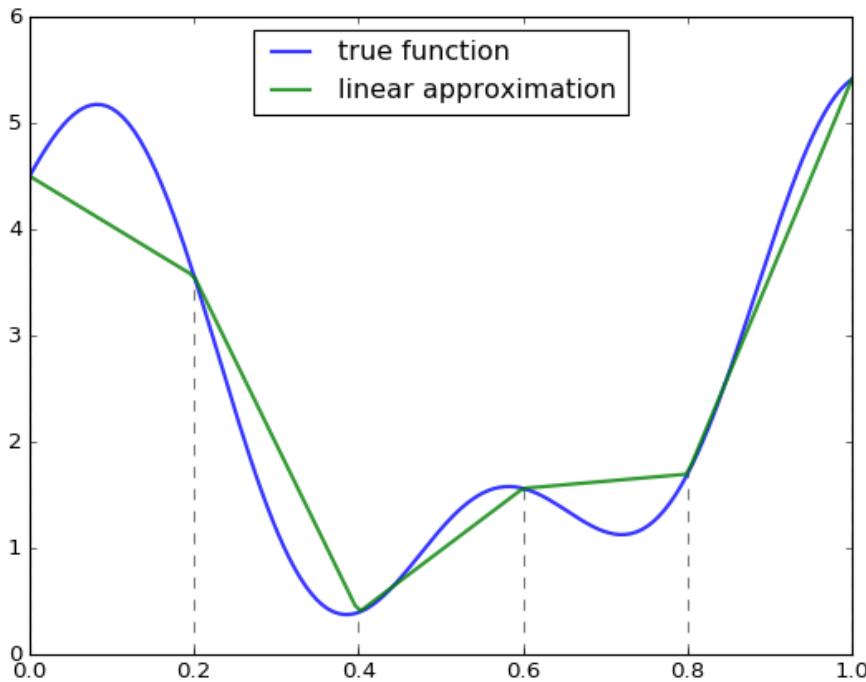
How should we go about step 2?

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  $0, 0.2, 0.4, \dots, 1$



Another advantage of piecewise linear interpolation is that it preserves useful shape properties such as monotonicity and concavity / convexity

**A First Pass Implementation** Let's now look at an implementation of fitted value function iteration using Python

In the example below,

- $f(k) = k^\alpha$  with  $\alpha = 0.65$
- $u(c) = \ln c$  and  $\beta = 0.95$

As is well-known (see [LS12], section 3.1.2), for this particular problem an exact analytical solution is available, with

$$v^*(k) = c_1 + c_2 \ln k \quad (2.88)$$

for

$$c_1 := \frac{\ln(1 - \alpha\beta)}{1 - \beta} + \frac{\ln(\alpha\beta)\alpha\beta}{(1 - \alpha\beta)(1 - \beta)} \quad \text{and} \quad c_2 := \frac{\alpha}{1 - \alpha\beta}$$

At this stage, our only aim is to see if we can replicate this solution numerically, using fitted value function iteration

Here's a first-pass solution, the details of which are explained *below*

The code can be found in file `optgrowth/optgrowth_v0.py` from the QuantEcon.applications repository

We repeat it here for convenience

```
"""
Filename: optgrowth_v0.py
Authors: John Stachurski and Tom Sargent

A first pass at solving the optimal growth problem via value function
iteration. A more general version is provided in optgrowth.py.

"""

from __future__ import division # Not needed for Python 3.x
import matplotlib.pyplot as plt
import numpy as np
from numpy import log
from scipy.optimize import fminbound
from scipy import interp

# Primitives and grid
alpha = 0.65
beta = 0.95
grid_max = 2
grid_size = 150
grid = np.linspace(1e-6, grid_max, grid_size)

# Exact solution
ab = alpha * beta
c1 = (log(1 - ab) + log(ab) * ab / (1 - ab)) / (1 - beta)
c2 = alpha / (1 - ab)

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

```

def bellman_operator(w):
    """
    The approximate Bellman operator, which computes and returns the updated
    value function  $T_w$  on the grid points.

    *  $w$  is a flat NumPy array with  $\text{len}(w) = \text{len}(\text{grid})$ 

    The vector  $w$  represents the value of the input function on the grid
    points.
    """
    # === Apply linear interpolation to  $w$  === #
    Aw = lambda x: interp(x, grid, w)

    # === set  $T_w[i]$  equal to  $\max_c \{ \log(c) + \beta w(f(k_i) - c) \}$  === #
    Tw = np.empty(grid_size)
    for i, k in enumerate(grid):
        objective = lambda c: - log(c) - beta * Aw(k**alpha - c)
        c_star = fminbound(objective, 1e-6, k**alpha)
        Tw[i] = - objective(c_star)

    return Tw

# === If file is run directly, not imported, produce figure === #
if __name__ == '__main__':
    w = 5 * log(grid) - 25  # An initial condition -- fairly arbitrary
    n = 35
    fig, ax = plt.subplots()
    ax.set_yscale(-40, -20)
    ax.set_xscale(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)
        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()

```

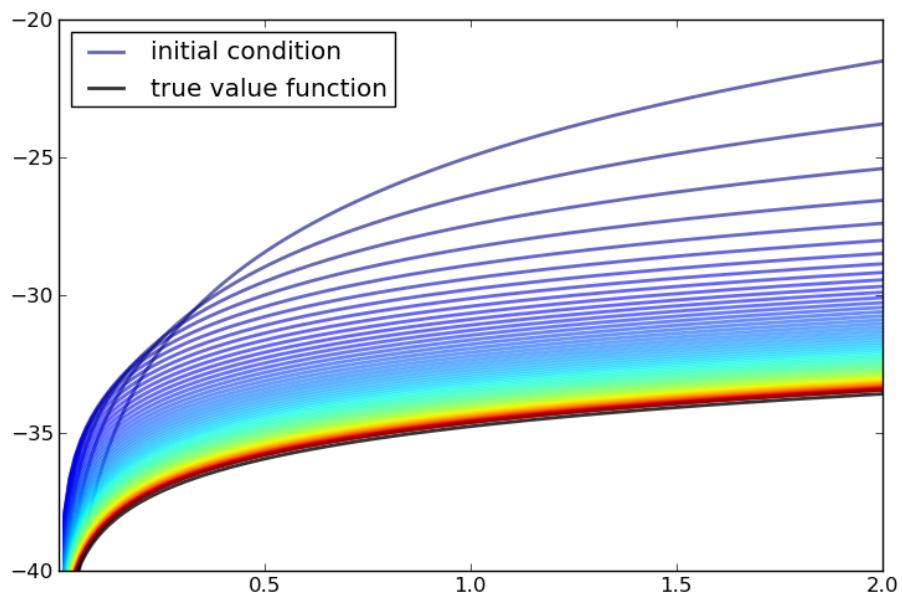
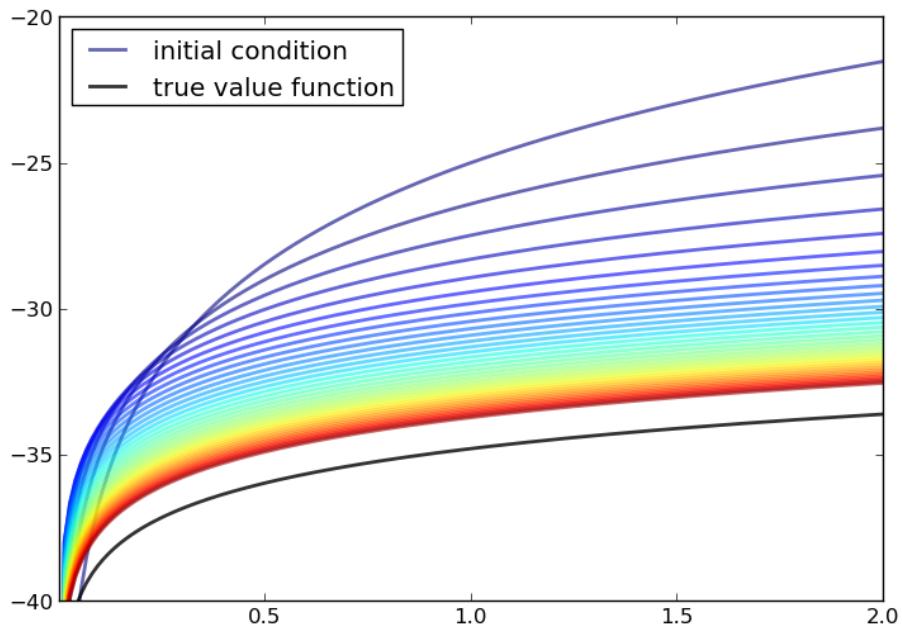
Running the code produces the following figure

The curves in this picture represent

1. the first 36 functions generated by the fitted value function iteration algorithm described above, with hotter colors given to higher iterates
2. the true value function as specified in (2.88), drawn in black

The sequence of iterates converges towards  $v^*$

If we increase  $n$  and run again we see further improvement — the next figure shows  $n = 75$



Incidentally, it is true that knowledge of the functional form of  $v^*$  for this model has influenced our choice of the initial condition

```
w = 5 * log(grid) - 25
```

In more realistic problems such information is not available, and convergence will probably take longer

**Comments on the Code** The function `bellman_operator` implements steps 2–3 of the fitted value function algorithm discussed *above*

Linear interpolation is performed by SciPy's `interp` function

Like the rest of SciPy's numerical solvers, `fminbound` minimizes its objective, so we use the identity  $\max_x f(x) = -\min_x -f(x)$  to solve (2.87)

The line `if __name__ == '__main__':` is common and operates as follows

- If the file is run as the result of an `import` statement in another file, the clause evaluates to `False`, and the code block is not executed
- If the file is run directly as a script, the clause evaluates to `True`, and the code block is executed

To see how this trick works, suppose we have a file in our current working directory called `test_file.py` that contains the single line

```
print(__name__)
```

Now consider the following, executed in IPython

```
In [1]: run test_file.py
__main__

In [2]: import test_file
test_file
```

Hopefully you can now see how it works

The benefit is that we can now import the functionality in `optgrowth_v0.py` without necessarily generating the figure

**The Policy Function** To compute an approximate optimal policy, we run the *fitted value function algorithm* until approximate convergence

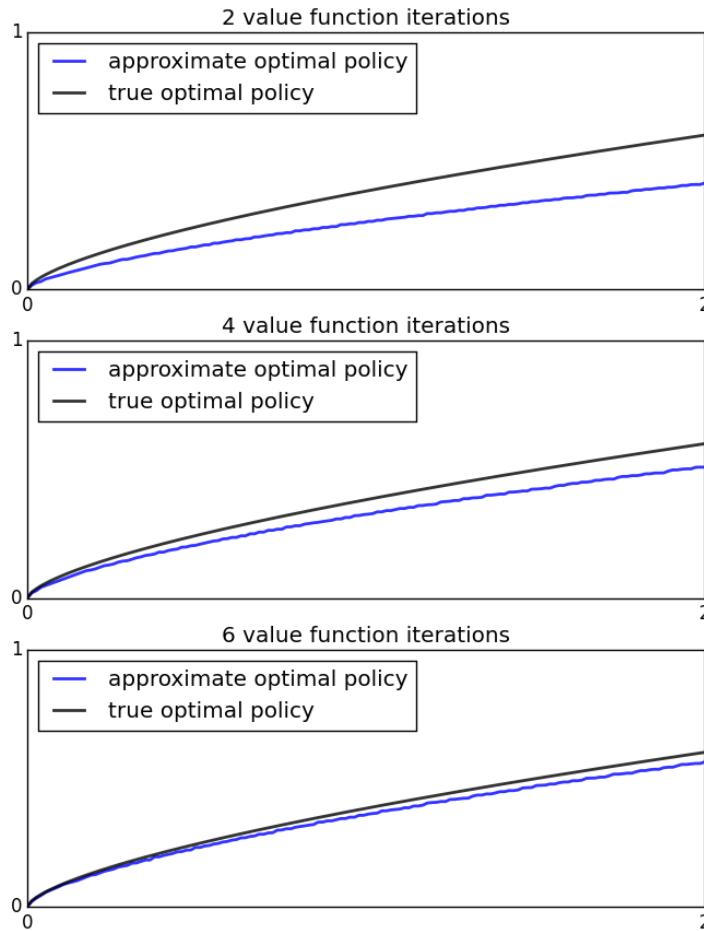
Taking the function so produced as an approximation to  $v^*$ , we then compute the (approximate)  $v^*$ -greedy policy

For this particular problem, the optimal consumption policy has the known analytical solution  $\sigma(k) = (1 - \alpha\beta)k^\alpha$

The next figure compares the numerical solution to this exact solution

In the three figures, the approximation to  $v^*$  is obtained by running the loop in the *fitted value function algorithm* 2, 4 and 6 times respectively

Even with as few as 6 iterates, the numerical result is quite close to the true policy



Exercise 1 asks you to reproduce this figure

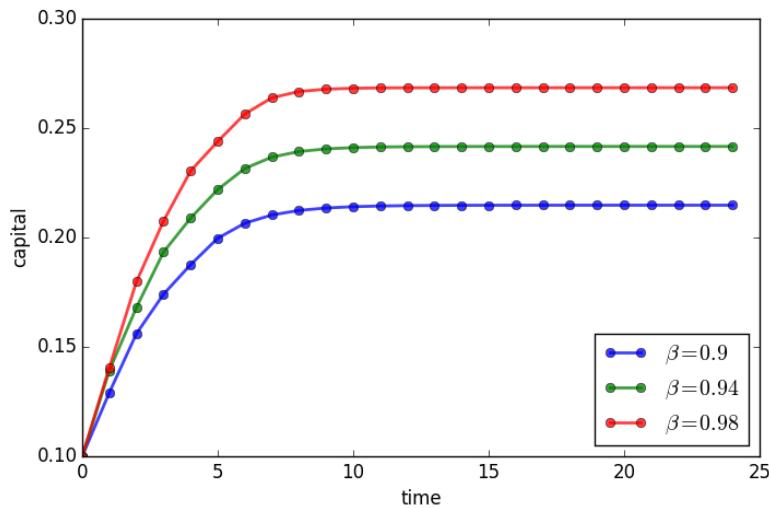
### Exercises

**Exercise 1** Replicate the optimal policy figure *shown above*

Use the same parameters and initial condition found in `optgrowth_v0.py`

**Exercise 2** Once an optimal consumption policy  $\sigma$  is given, the dynamics for the capital stock follows (2.82)

The next figure shows the first 25 elements of this sequence for three different discount factors (and hence three different policies)



In each sequence, the initial condition is  $k_0 = 0.1$

The discount factors are `discount_factors = (0.9, 0.94, 0.98)`

Otherwise, the parameters and primitives are the same as found in `optgrowth_v0.py`

Replicate the figure

## Solutions

[Solution notebook](#)

# Optimal Growth Part II: Adding Some Bling

## Contents

- *Optimal Growth Part II: Adding Some Bling*
  - [Overview](#)
  - [Adding a Shock](#)
  - [Implementation](#)
  - [Exercises](#)
  - [Solutions](#)

## Overview

In the [last lecture](#) we

- studied a simple version of the one sector optimal growth model
- put together some fairly basic code to solve for the optimal policy

In this lecture we're going to make some improvements

In terms of the model, we're going to add a random component to production

In terms of the code, we're going to

- generalize the *bellman\_operator* function to handle other models
- refactor (reorganize) the code to make it easier to interact with
- make use of some *QuantEcon* utilities to run the iteration
- make use of caching of results to speed up computations

Caching uses the *joblib* library, which you need to install to run the code in this lecture

This can be done at a shell prompt by typing

```
pip install joblib
```

### **Adding a Shock**

The first step is to modify the problem to include a production shock

The shock sequence will be denoted  $\{\xi_t\}$  and assumed to be IID for simplicity

Many treatments include  $\xi_t$  as one of the state variables but this can be avoided in the IID case if we choose the timing appropriately

**Timing** The timing we adopt is

1. At the start of period  $t$ , current output  $y_t$  is observed
2. Consumption  $c_t$  is chosen, and the remainder  $y_t - c_t$  is used as productive capital
3. The shock  $\xi_{t+1}$  is realized
4. Production takes place, yielding output  $y_{t+1} = f(y_t - c_t)\xi_{t+1}$

Now  $t$  increments and the process repeats

Comments:

- The “current” shock  $\xi_{t+1}$  has subscript  $t + 1$  because it is not in the time  $t$  information set
- The production function  $f$  is assumed to be continuous
- The shock is multiplicative by assumption — this is not the only possibility
- Depreciation is not made explicit but can be incorporated into the production function

**The Optimization Problem** Taking  $y_0$  as given, the agent wishes to maximize

$$\mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(c_t) \right] \quad (2.89)$$

subject to

$$y_{t+1} = f(y_t - c_t) \xi_{t+1} \quad \text{and} \quad 0 \leq c_t \leq y_t \quad \text{for all } t$$

For interpretation see the [first optimal growth lecture](#)

The difference is in the timing, as discussed above, and the addition of the expectation term  $\mathbb{E}$

Note also that  $y_t$  is the state variable rather than  $k_t$

We seek again a Markov policy, which is a function  $\sigma: \mathbb{R}_+ \rightarrow \mathbb{R}_+$  that maps states to actions:

$$c_t = \sigma(y_t) \quad \text{for all } t$$

Analogous to the [first optimal growth lecture](#), we will call  $\sigma$  a *feasible consumption policy* if it satisfies

$$0 \leq \sigma(y) \leq y \quad \text{for all } y \in \mathbb{R}_+ \quad (2.90)$$

The set of all such policies will be denoted by  $\Sigma$

Each feasible policy  $\sigma$  determines a Markov process  $\{y_t\}$  via

$$y_{t+1} = f(y_t - \sigma(y_t)) \xi_{t+1}, \quad y_0 \text{ given} \quad (2.91)$$

We insert this process into

$$\mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(\sigma(y_t)) \right] \quad (2.92)$$

to obtain the total expected present value of following policy  $\sigma$  forever, given initial income  $y_0$

The aim is to select a policy that makes this number as large as possible

The next section covers these ideas more formally

**Optimality and the Bellman Operator** The *policy value function*  $v_\sigma$  associated with a given policy  $\sigma$  is

$$v_\sigma(y_0) := \mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(\sigma(y_t)) \right] \quad (2.93)$$

when  $\{y_t\}$  is given by (2.91)

The *value function* is then defined as

$$v^*(y_0) := \sup_{\sigma \in \Sigma} v_\sigma(y_0) \quad (2.94)$$

The *Bellman equation* for this problem takes the form

$$v^*(y) = \max_{0 \leq c \leq y} \{u(c) + \beta \mathbb{E} v^*(f(y - c) \xi)\} \quad \text{for all } y \in \mathbb{R}_+ \quad (2.95)$$

Analogous to the Bellman equation in the [first optimal growth lecture](#), it states that maximal value from a given state can be obtained by trading off

- current reward from a given action vs
- expected discounted future value of the state resulting from that action

The corresponding Bellman operator  $w \mapsto Tw$  is

$$Tw(y) := \max_{0 \leq c \leq y} \{u(c) + \beta \mathbb{E}w(f(y - c)\xi)\} \quad (2.96)$$

As in the [first optimal growth lecture](#), if

- $u$  and  $f$  are continuous, and
- $u$  is bounded and  $\beta < 1$ ,

then we can compute  $v^*$  by iterating with  $T$  from any initial continuous bounded function on the state space

For background and theory see, for example,

- [\[MZ75\]](#)
- lemma 10.1.20 of [EDTC](#)

Even if  $u$  is unbounded, we can often find theoretical justification for this kind of iterative procedure (see, e.g., lemma 12.2.25 of [EDTC](#))

**Greedy and Optimal Policies** A policy  $\sigma \in \Sigma$  is called *optimal* if it attains the supremum in (2.94) for all  $y_0 \in \mathbb{R}_+$

Given a continuous function  $w$  on  $\mathbb{R}_+$ , we say that  $\sigma \in \Sigma$  is  $w$ -greedy if  $\sigma(y)$  is a solution to

$$\max_{0 \leq c \leq y} \{u(c) + \beta \mathbb{E}v^*(f(y - c)\xi)\} \quad (2.97)$$

for every  $y \in \mathbb{R}_+$

A feasible policy is optimal if and only if it is  $v^*$ -greedy policy (see, e.g., theorem 10.1.11 of [EDTC](#))

Hence, once we have a good approximation to  $v^*$ , we can compute the (approximately) optimal policy by taking the greedy policy

### Implementation

To solve the model we'll write code that

- generalizes our [earlier effort](#)
- is more reusable and modular
- is easier to interact with in repeated simulations

**The Bellman Operator** As in the first optimal growth lecture,

- iteration involves function approximation
- function approximation will be carried out by piecewise linear interpolation

We take the *bellman\_operator* function from that lecture and rewrite it as follows

```
"""
Filename: optgrowth.py
Authors: John Stachurski, Thomas Sargent

Solving the optimal growth problem via value function iteration. The model is
described in

    http://quant-econ.net/py/optgrowth_2.html
"""

import numpy as np
from scipy.optimize import fminbound
from scipy import interp

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
    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).
    beta : scalar
        The discount factor
    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: interp(x, grid, w)
```

```

# == Initialize Tw if necessary == #
if Tw is None:
    Tw = np.empty(len(w))

if compute_policy:
    sigma = np.empty(len(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

```

This code is from the file `optgrowth_2/optgrowth.py` from the `QuantEcon.applications` repository

The arguments to `bellman_operator` are described in the docstring to the function

Comments:

- We can now pass in any production and utility function
- The expectation in (2.96) is computed via Monte Carlo, using the approximation

$$\mathbb{E}w(f(y - c)\xi) \approx \frac{1}{n} \sum_{i=1}^n w(f(y - c)\xi_i)$$

where  $\{\xi_i\}_{i=1}^n$  are IID draws from the distribution of the shock

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])

**Solving Specific Models** Now we want to write some code that uses the Bellman operator to solve specific models

The first model we'll try out is one with log utility and Cobb–Douglas production, similar to the first optimal growth lecture

This model is simple and has a closed form solution but is useful as a test case

We assume that shocks are lognormal, with  $\ln \xi_t \sim N(\mu, \sigma^2)$

We present the code first, which is file `optgrowth_2/log_linear_growth_model.py` from the QuantEcon.applications repository

Explanations are given after the code

```

"""
Filename: log_linear_growth_model.py
Authors: John Stachurski, Thomas Sargent

The log linear growth model, wrapped as classes. For use with the
optgrowth.py module.
"""

import numpy as np
import matplotlib.pyplot as plt
from optgrowth import bellman_operator
from quantecon import compute_fixed_point
from joblib import Memory

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

@memory.cache
def compute_value_function_cached(grid, beta, alpha, shocks):
    """
    Compute the value function by iterating on the Bellman operator.
    The work is done by QuantEcon's compute_fixed_point function.
    """
    Tw = np.empty(len(grid))
    initial_w = 5 * np.log(grid) - 25

    v_star = compute_fixed_point(bellman_operator,
                                  initial_w,
                                  1e-4, # error_tol
                                  100, # max_iter
                                  True, # verbose
                                  5, # print_skip
                                  grid,
                                  beta,
                                  np.log,
                                  lambda k: k**alpha,
                                  shocks,
                                  Tw=Tw,
                                  compute_policy=False)
    return v_star

class LogLinearGrowthModel:
    """
    Stores parameters and computes solutions for the basic log utility / Cobb
    Douglas production growth model. Shocks are lognormal.
    """

    def __init__(self,
                 alpha=0.65,          # Productivity parameter
                 beta=0.95,           # Discount factor

```

```

        mu=1,                      # First parameter in lognorm(mu, sigma)
        sigma=0.1,                  # Second parameter in lognorm(mu, sigma)
        grid_max=8,
        grid_size=150):

    self.alpha, self.beta, self.mu, self.sigma = alpha, beta, mu, sigma
    self.grid = np.linspace(1e-6, grid_max, grid_size)
    self.shocks = np.exp(mu + sigma * np.random.randn(250))

def compute_value_function(self, show_plot=False):
    """
    Calls compute_value_function_cached and optionally adds a plot.
    """
    v_star = compute_value_function_cached(self.grid,
                                            self.beta,
                                            self.alpha,
                                            self.shocks)

    if show_plot:
        fig, ax = plt.subplots()
        ax.plot(self.grid, v_star, lw=2, alpha=0.6, label='value function')
        ax.legend(loc='lower right')
        plt.show()

    return v_star

def compute_greedy(self, w=None, show_plot=False):
    """
    Compute the w-greedy policy on the grid points given w
    (the value of the input function on grid points). If w is not
    supplied, use the approximate optimal value function.
    """
    if w is None:
        w = self.compute_value_function()

    Tw, sigma = bellman_operator(w,
                                 self.grid,
                                 self.beta,
                                 np.log,
                                 lambda k: k**self.alpha,
                                 self.shocks,
                                 compute_policy=True)

    if show_plot:
        fig, ax = plt.subplots()
        ax.plot(self.grid, sigma, lw=2, alpha=0.6, label='approximate policy function')
        cstar = (1 - self.alpha * self.beta) * self.grid
        ax.plot(self.grid, cstar, lw=2, alpha=0.6, label='true policy function')
        ax.legend(loc='upper left')
        plt.show()

    return sigma

```

Let's discuss some of the interesting features of the code

**Computing the Value Function** The function `compute_value_function_cached` performs iteration to convergence for this particular model

In turn, it uses the function `compute_fixed_point` from `quantecon` to control iteration

This function

- provides a generic interface to the kinds of iteration problems routinely encountered in economic modeling
- avoids allocating additional memory at each step, unlike our code from the [first optimal growth lecture](#)

**Caching** Notice the three lines

```
from joblib import Memory
memory = Memory(cachedir='./joblib_cache')

@memory.cache
```

We are using the `joblib` library to cache the result of calling `compute_value_function_cached` 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

Now if we call the function twice with the same set of parameters, the result will be returned almost instantaneously

**Models as Classes** The parameters and the methods that we actually interact with are wrapped in a class called `LogLinearGrowthModel`

This keeps our variables organized in one self-contained object

It also makes it easier to evaluate endogenous entities like value functions across a range of parameters

For example, to compute the value function at  $\alpha \in \{0.5, 0.55, 0.6, 0.65\}$  we could use the following

```
val_functions = []
alphas = [0.5, 0.55, 0.6, 0.65]
for alpha in alphas:
    gm = LogLinearGrowthModel(alpha=alpha)
    vstar = gm.compute_value_function()
    val_functions.append(vstar)
```

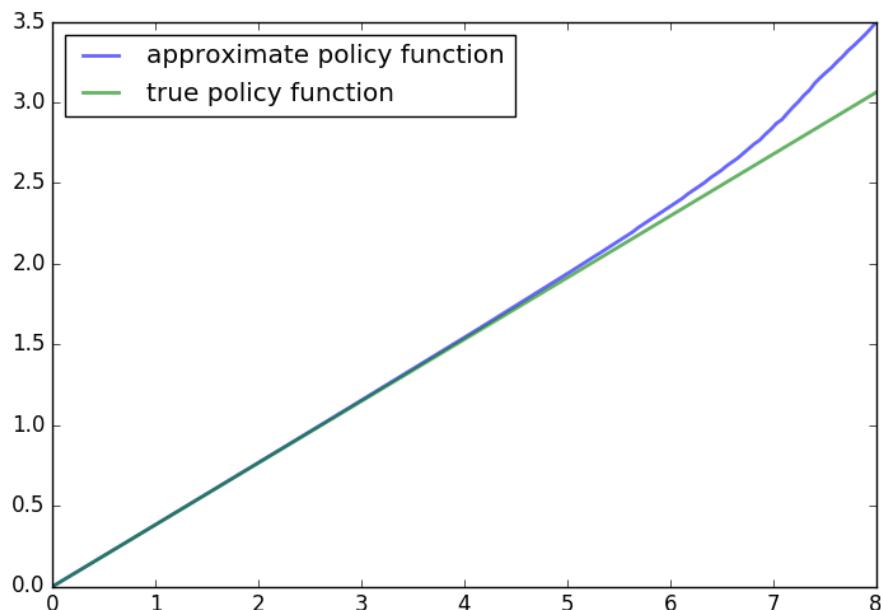
This client code is simple and neat, and hence less error prone than client code involving complex function calls

**A Test** For this growth model it is known (see, e.g., chapter 1 of EDTC) that the optimal policy for consumption is  $\sigma(y) = (1 - \alpha\beta)y$

Let's see if we match this when we run

```
In [1]: run log_linear_growth_model.py
In [2]: lg = LogLinearGrowthModel()
In [3]: lg.compute_greedy(show_plot=True)
```

The resulting figure looks as follows



The approximation is good apart from close to the right hand boundary

Why is the approximation poor at the upper end of the grid?

This is caused by a combination of the function approximation step and the shock component

In particular, with the *interp* routine that we are using, evaluation at points larger than any grid point returns the value at the right-most grid point

Points larger than any grid point are encountered in the Bellman operator at the step where we take expectations

One solution is to take the grid larger than where we wish to compute the policy

Another is to tweak the *interp* routine, although we won't pursue that idea here

## Exercises

Coming soon.

## Solutions

Coming soon.

# 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, although we won't pursue the deeper connections in 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

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \quad t = 0, 1, 2, \dots \quad (2.98)$$

Here

- $u_t$  is a “control” vector, incorporating choices available to a decision maker confronting the current state  $x_t$
- $\{w_t\}$  is an uncorrelated zero mean shock process satisfying  $\mathbb{E}w_tw_t' = I$ , where the right-hand side is the identity matrix

Regarding the dimensions

- $x_t$  is  $n \times 1$ ,  $A$  is  $n \times n$
- $u_t$  is  $k \times 1$ ,  $B$  is  $n \times k$
- $w_t$  is  $j \times 1$ ,  $C$  is  $n \times j$

**Example 1** Consider a household budget constraint given by

$$a_{t+1} + c_t = (1 + r)a_t + y_t$$

Here  $a_t$  is assets,  $r$  is a fixed interest rate,  $c_t$  is current consumption, and  $y_t$  is current non-financial income

If we suppose that  $\{y_t\}$  is 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 (2.98), with assets being the state and consumption being the control

**Example 2** One unrealistic feature of the previous model is that non-financial income has a zero mean and is often negative

This can easily be overcome by adding a sufficiently large mean

Hence in this example we take  $y_t = \sigma w_{t+1} + \mu$  for some positive real number  $\mu$

Another alteration that'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

$$a_{t+1} = (1+r)a_t - u_t - \bar{c} + \sigma w_{t+1} + \mu \quad (2.99)$$

How can we write this new system in the form of equation (2.98)?

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

$$\begin{pmatrix} a_{t+1} \\ 1 \end{pmatrix} = \begin{pmatrix} 1+r & -\bar{c} + \mu \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_t \\ 1 \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix} u_t + \begin{pmatrix} \sigma \\ 0 \end{pmatrix} w_{t+1} \quad (2.100)$$

then the first row is equivalent to (2.99)

Moreover, the model is now linear, and can be written in the form of (2.98) by setting

$$x_t := \begin{pmatrix} a_t \\ 1 \end{pmatrix}, \quad A := \begin{pmatrix} 1+r & -\bar{c} + \mu \\ 0 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \end{pmatrix} \quad (2.101)$$

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

$$x_t' R x_t + u_t' Q u_t \quad (2.102)$$

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 (2.102) 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  $\{u_0, \dots, u_{T-1}\}$  to minimize the objective

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (x_t' R x_t + u_t' Q u_t) + \beta^T x_T' R_f x_T \right\} \quad (2.103)$$

subject to the law of motion (2.98) and initial state  $x_0$

The new objects introduced here are  $\beta$  and the matrix  $R_f$

The scalar  $\beta$  is the discount factor, while  $x' R_f x$  gives terminal loss associated with state  $x$

Comments:

- We assume  $R_f$  to be  $n \times n$ , symmetric and nonnegative definite
- We allow  $\beta = 1$ , and hence include the undiscounted case
- $x_0$  may itself be random, in which case we require it to be independent of the shock sequence  $w_1, \dots, w_T$

**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  $\{u_0, \dots, u_{T-1}\}$

When choosing these controls, the decision maker is permitted to take into account the effects of the shocks  $\{w_1, \dots, w_T\}$  on the system

However, it is typically assumed — and will be assumed here — that the time- $t$  control  $u_t$  can be made with knowledge of past and present shocks only

The fancy [measure-theoretic](#) way of saying this is that  $u_t$  must be measurable with respect to the  $\sigma$ -algebra generated by  $x_0, w_1, w_2, \dots, w_t$

This is in fact equivalent to stating that  $u_t$  can be written in the form  $u_t = g_t(x_0, w_1, w_2, \dots, w_t)$  for some Borel measurable function  $g_t$

(Just about every function that's useful for applications is Borel measurable, so, for the purposes of intuition, you can read that last phrase as "for some function  $g_t$ ")

Now note that  $x_t$  will ultimately depend on the realizations of  $x_0, w_1, w_2, \dots, w_t$

In fact it turns out that  $x_t$  summarizes all the information about these historical shocks that the decision maker needs to set controls optimally

More precisely, it can be shown that any optimal control  $u_t$  can always be written as a function of the current state alone

Hence in what follows we restrict attention to control policies (i.e., functions) of the form  $u_t = g_t(x_t)$

Actually, the preceding discussion applies to all standard dynamic programming problems

What's special about the LQ case is that — as we shall soon see — the optimal  $u_t$  turns out to be a linear function of  $x_t$

**Solution** To solve the finite horizon LQ problem we can use a dynamic programming strategy based on backwards induction that is conceptually similar to the approach adopted in [this lecture](#)

For reasons that will soon become clear, we first introduce the notation  $J_T(x) := x' R_f x$

Now consider the problem of the decision maker in the second to last period

In particular, let the time be  $T - 1$ , and suppose that the state is  $x_{T-1}$

The decision maker must trade off current and (discounted) final losses, and hence solves

$$\min_u \{x'_{T-1} Rx_{T-1} + u' Qu + \beta \mathbb{E} J_T(Ax_{T-1} + Bu + Cw_T)\}$$

At this stage, it is convenient to define the function

$$J_{T-1}(x) := \min_u \{x' Rx + u' Qu + \beta \mathbb{E} J_T(Ax + Bu + Cw_T)\} \quad (2.104)$$

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

$$\min_u \{x'_{T-2} Rx_{T-2} + u' Qu + \beta \mathbb{E} J_{T-1}(Ax_{T-2} + Bu + Cw_{T-1})\}$$

Letting

$$J_{T-2}(x) := \min_u \{x' Rx + u' Qu + \beta \mathbb{E} J_{T-1}(Ax + Bu + Cw_{T-1})\}$$

the pattern for backwards induction is now clear

In particular, we define a sequence of value functions  $\{J_0, \dots, J_T\}$  via

$$J_{t-1}(x) = \min_u \{x' Rx + u' Qu + \beta \mathbb{E} J_t(Ax + Bu + Cw_t)\} \quad \text{and} \quad J_T(x) = x' R_f x$$

The first equality is the Bellman equation from dynamic programming theory specialized to the finite horizon LQ problem

Now that we have  $\{J_0, \dots, J_T\}$ , we can obtain the optimal controls

As a first step, let's find out what the value functions look like

It turns out that every  $J_t$  has the form  $J_t(x) = x' P_t x + d_t$  where  $P_t$  is a  $n \times n$  matrix and  $d_t$  is a constant

We can show this by induction, starting from  $P_T := R_f$  and  $d_T = 0$

Using this notation, (2.104) becomes

$$J_{T-1}(x) := \min_u \{x' Rx + u' Qu + \beta \mathbb{E} (Ax + Bu + Cw_T)' P_T (Ax + Bu + Cw_T)\} \quad (2.105)$$

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

$$u = -(Q + \beta B' P_T B)^{-1} \beta B' P_T A x \quad (2.106)$$

Plugging this back into (2.105) and rearranging yields

$$J_{T-1}(x) := x' P_{T-1} x + d_{T-1}$$

where

$$P_{T-1} := R - \beta^2 A' P_T B (Q + \beta B' P_T B)^{-1} B' P_T A + \beta A' P_T A \quad (2.107)$$

and

$$d_{T-1} := \beta \operatorname{trace}(C' P_T C) \quad (2.108)$$

(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

$$P_{t-1} := R - \beta^2 A' P_t B (Q + \beta B' P_t B)^{-1} B' P_t A + \beta A' P_t A \quad \text{with} \quad P_T = R_f \quad (2.109)$$

and

$$d_{t-1} := \beta(d_t + \operatorname{trace}(C' P_t C)) \quad \text{with} \quad d_T = 0 \quad (2.110)$$

Recalling (2.106), the minimizers from these backward steps are

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B' P_{t+1} B)^{-1} \beta B' P_{t+1} A \quad (2.111)$$

These are the linear optimal control policies we *discussed above*

In particular, the sequence of controls given by (2.111) and (2.98) solves our finite horizon LQ problem

Rephrasing this more precisely, the sequence  $u_0, \dots, u_{T-1}$  given by

$$u_t = -F_t x_t \quad \text{with} \quad x_{t+1} = (A - BF_t)x_t + Cw_{t+1} \quad (2.112)$$

for  $t = 0, \dots, T-1$  attains the minimum of (2.103) 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 many others built models based on a consumer's preference for a stable 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

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (c_t - \bar{c})^2 + \beta^T q a_T^2 \right\} \quad (2.113)$$

subject to the sequence of budget constraints  $a_{t+1} = (1+r)a_t - c_t + y_t$ ,  $t \geq 0$

Here  $q$  is a large positive constant, the role of which is to induce the consumer to target zero debt at the end of her life

(Without such a constraint, the optimal choice is to choose  $c_t = \bar{c}$  in each period, letting assets adjust accordingly)

As before we set  $y_t = \sigma w_{t+1} + \mu$  and  $u_t := c_t - \bar{c}$ , after which the constraint can be written as in (2.99)

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 (2.101)

To match with this state and control, the objective function (2.113) can be written in the form of (2.103) 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 (2.109) and (2.111)

After generating shocks  $w_1, \dots, w_T$ , the dynamics for assets and consumption can be simulated via (2.112)

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

$$z_t := \sum_{j=0}^t \sigma w_j$$

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)

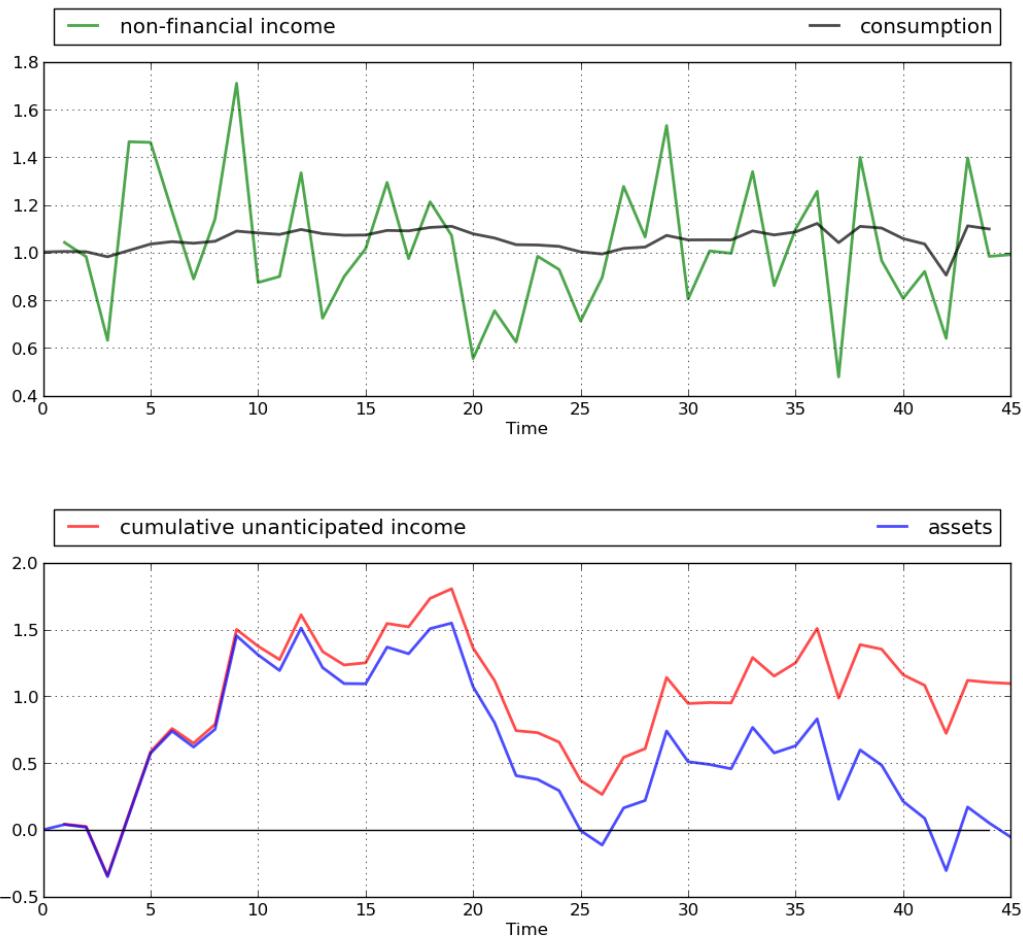
These results are relatively robust to changes in parameters

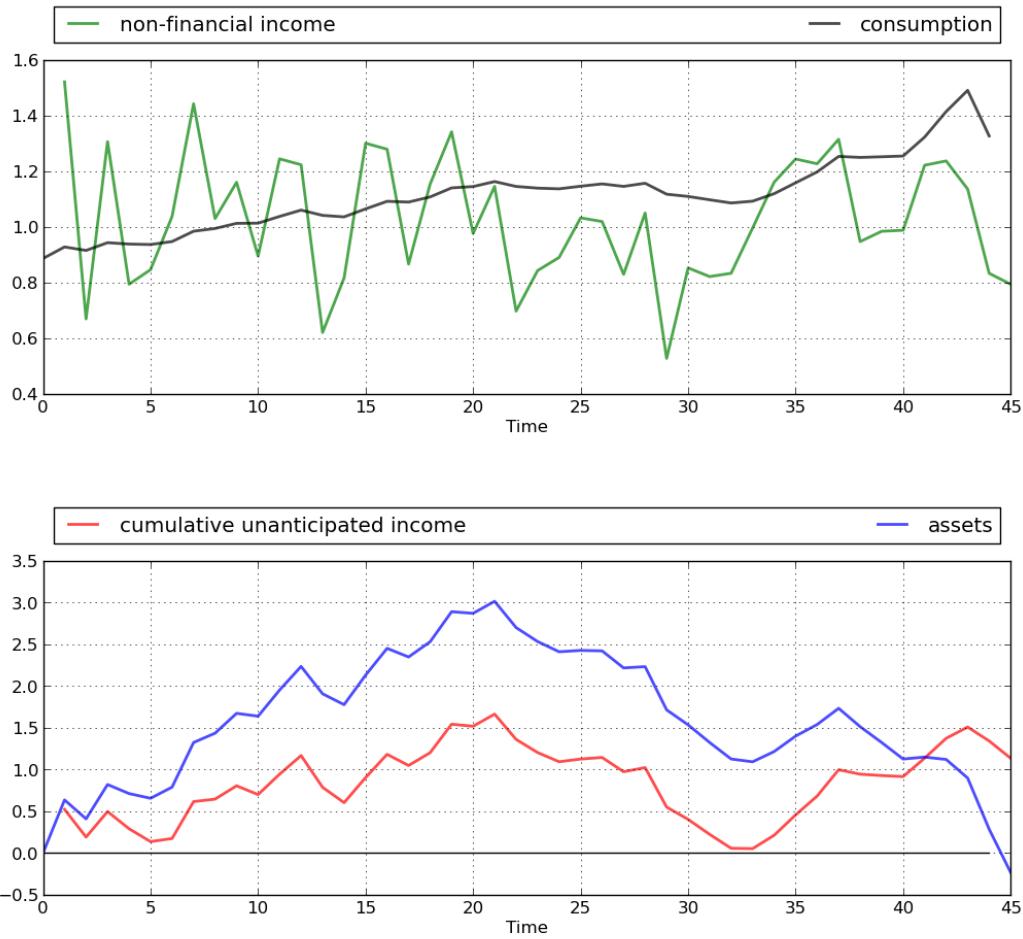
For example, let's increase  $\beta$  from  $1/(1+r) \approx 0.952$  to 0.96 while keeping other parameters fixed

This consumer is slightly more patient than the last one, and hence puts relatively more weight on later consumption values

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

**Nonstationary 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 nonstationary models from within our implementation 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

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (x_t' R x_t + u_t' Q u_t + 2u_t' N x_t) + \beta^T x_T' R_f x_T \right\} \quad (2.114)$$

Our results extend to this case in a straightforward way

The sequence  $\{P_t\}$  from (2.109) becomes

$$P_{t-1} := R - (\beta B' P_t A + N)' (Q + \beta B' P_t B)^{-1} (\beta B' P_t A + N) + \beta A' P_t A \quad \text{with} \quad P_T = R_f \quad (2.115)$$

The policies in (2.111) are modified to

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B' P_{t+1} B)^{-1} (\beta B' P_{t+1} A + N) \quad (2.116)$$

The sequence  $\{d_t\}$  is unchanged from (2.110)

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

$$\mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t (x_t' R x_t + u_t' Q u_t + 2u_t' N x_t) \right\} \quad (2.117)$$

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  $u_t = -Fx_t$  for all  $t$

This stationarity 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 given by the fixed point of (2.115)

Equivalently, it is the solution  $P$  to the [discrete time algebraic Riccati equation](#)

$$P := R - (\beta B'PA + N)'(Q + \beta B'PB)^{-1}(\beta B'PA + N) + \beta A'PA \quad (2.118)$$

Equation (2.118) is also called the *LQ Bellman equation*, and the map that sends a given  $P$  into the right-hand side of (2.118) is called the *LQ Bellman operator*

The stationary optimal policy for this model is

$$u = -Fx \quad \text{where} \quad F := (Q + \beta B'PB)^{-1}(\beta B'PA + N) \quad (2.119)$$

The sequence  $\{d_t\}$  from (2.110) is replaced by the constant value

$$d := \text{trace}(C'PC) \frac{\beta}{1 - \beta} \quad (2.120)$$

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 (2.119) or (2.116)

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](#) but we repeat it here for convenience

```

"""
Filename: lqcontrol.py

Authors: Thomas J. Sargent, John Stachurski

Provides a class called LQ for solving linear quadratic control
problems.

"""

from textwrap import dedent
import numpy as np
from numpy import dot
from scipy.linalg import solve
from .matrix_eqn import solve_discrete_riccati


class LQ(object):
    """
    This class is for analyzing linear quadratic optimal control
    problems of either the infinite horizon form

        min E sum_{t=0}^{\infty} beta^t r(x_t, u_t)

    with

        r(x_t, u_t) := x_t' R x_t + u_t' Q u_t + 2 u_t' N x_t

    or the finite horizon form

        min E sum_{t=0}^{T-1} beta^t r(x_t, u_t) + beta^T x_T' R_f x_T

    Both are minimized subject to the law of motion

        x_{t+1} = A x_t + B u_t + C w_{t+1}

    Here x is n x 1, u is k x 1, w is j x 1 and the matrices are
    conformable for these dimensions. The sequence {w_t} is assumed to
    be white noise, with zero mean and E w_t w_t' = I, the j x j
    identity.

    If C is not supplied as a parameter, the model is assumed to be
    deterministic (and C is set to a zero matrix of appropriate
    dimension).

    For this model, the time t value (i.e., cost-to-go) function V_t
    takes the form

        x' P_T x + d_T

    and the optimal policy is of the form u_T = -F_T x_T. In
    the infinite horizon case, V, P, d and F are all stationary.

    Parameters

```

```

-----
Q : array_like(float)
    Q is the payoff(or cost) matrix that corresponds with the
    control variable u and is k x k. Should be symmetric and
    nonnegative definite
R : array_like(float)
    R is the payoff(or cost) matrix that corresponds with the
    state variable x and is n x n. Should be symmetric and
    non-negative definite
N : array_like(float)
    N is the cross product term in the payoff, as above. It should
    be k x n.
A : array_like(float)
    A is part of the state transition as described above. It should
    be n x n
B : array_like(float)
    B is part of the state transition as described above. It should
    be n x k
C : array_like(float), optional(default=None)
    C is part of the state transition as described above and
    corresponds to the random variable today. If the model is
    deterministic then C should take default value of None
beta : scalar(float), optional(default=1)
    beta is the discount parameter
T : scalar(int), optional(default=None)
    T is the number of periods in a finite horizon problem.
Rf : array_like(float), optional(default=None)
    Rf is the final (in a finite horizon model) payoff(or cost)
    matrix that corresponds with the control variable u and is n x
    n. Should be symmetric and non-negative definite

```

**Attributes**

```

-----
Q, R, N, A, B, C, beta, T, Rf : see Parameters
P : array_like(float)
    P is part of the value function representation of  $V(x) = x'Px + d$ 
d : array_like(float)
    d is part of the value function representation of  $V(x) = x'Px + d$ 
F : array_like(float)
    F is the policy rule that determines the choice of control in
    each period.
k, n, j : scalar(int)
    The dimensions of the matrices as presented above

```

```
"""
```

```

def __init__(self, Q, R, A, B, C=None, N=None, beta=1, T=None, Rf=None):
    # == Make sure all matrices can be treated as 2D arrays == #
    converter = lambda X: np.atleast_2d(np.asarray(X, dtype='float'))
    self.A, self.B, self.Q, self.R, self.N = list(map(converter,
                                                    (A, B, Q, R, N)))
    # == Record dimensions == #

```

```

    self.k, self.n = self.Q.shape[0], self.R.shape[0]

    self.beta = beta

    if C is None:
        # == If C not given, then model is deterministic. Set C=0. == #
        self.j = 1
        self.C = np.zeros((self.n, self.j))
    else:
        self.C = converter(C)
        self.j = self.C.shape[1]

    if N is None:
        # == No cross product term in payoff. Set N=0. == #
        self.N = np.zeros((self.k, self.n))

    if T:
        # == Model is finite horizon == #
        self.T = T
        self.Rf = np.asarray(Rf, dtype='float')
        self.P = self.Rf
        self.d = 0
    else:
        self.P = None
        self.d = None
        self.T = None

    self.F = None

def __repr__(self):
    return self.__str__()

def __str__(self):
    m = """\
    Linear Quadratic control system
    - beta (discount parameter)      : {b}
    - T (time horizon)              : {t}
    - n (number of state variables) : {n}
    - k (number of control variables) : {k}
    - j (number of shocks)          : {j}
    """
    t = "infinite" if self.T is None else self.T
    return dedent(m.format(b=self.beta, n=self.n, k=self.k, j=self.j,
                           t=t))

def update_values(self):
    """
    This method is for updating in the finite horizon case. It
    shifts the current value function

    
$$V_t(x) = x' P_t x + d_t$$


    and the optimal policy  $F_t$  one step *back* in time,

```

```

replacing the pair  $P_t$  and  $d_t$  with
 $P_{t-1}$  and  $d_{t-1}$ , and  $F_t$  with
 $F_{t-1}$ 

"""

# === Simplify notation === #
Q, R, A, B, N, C = self.Q, self.R, self.A, self.B, self.N, self.C
P, d = self.P, self.d
# == Some useful matrices ==
S1 = Q + self.beta * dot(B.T, dot(P, B))
S2 = self.beta * dot(B.T, dot(P, A)) + N
S3 = self.beta * dot(A.T, dot(P, A))
# == Compute F as  $(Q + B'PB)^{-1} (beta B'PA + N)$  ==
self.F = solve(S1, S2)
# === Shift P back in time one step ==
new_P = R - dot(S2.T, self.F) + S3
# == Recalling that trace(AB) = trace(BA) ==
new_d = self.beta * (d + np.trace(dot(P, dot(C, C.T))))
# == Set new state ==
self.P, self.d = new_P, new_d

def stationary_values(self):
    """
    Computes the matrix P and scalar d that represent the value
    function

     $V(x) = x' P x + d$ 

    in the infinite horizon case. Also computes the control matrix
    F from  $u = -Fx$ 

    Returns
    ----
    P : array_like(float)
        P is part of the value function representation of
         $V(x) = xPx + d$ 
    F : array_like(float)
        F is the policy rule that determines the choice of control
        in each period.
    d : array_like(float)
        d is part of the value function representation of
         $V(x) = xPx + d$ 

    """
    # === simplify notation === #
    Q, R, A, B, N, C = self.Q, self.R, self.A, self.B, self.N, self.C

    # == solve Riccati equation, obtain P ==
    A0, B0 = np.sqrt(self.beta) * A, np.sqrt(self.beta) * B
    P = solve_discrete_riccati(A0, B0, R, Q, N)

    # == Compute F ==
    S1 = Q + self.beta * dot(B.T, dot(P, B))

```

```

S2 = self.beta * dot(B.T, dot(P, A)) + N
F = solve(S1, S2)

# == Compute d == #
d = self.beta * np.trace(dot(P, dot(C, C.T))) / (1 - self.beta)

# == Bind states and return values == #
self.P, self.F, self.d = P, F, d

return P, F, d

def compute_sequence(self, x0, ts_length=None):
    """
    Compute and return the optimal state and control sequences
    x_0, ..., x_T and u_0, ..., u_T under the
    assumption that {w_t} is iid and N(0, 1).

    Parameters
    ======
    x0 : array_like(float)
        The initial state, a vector of length n

    ts_length : scalar(int)
        Length of the simulation -- defaults to T in finite case

    Returns
    ======
    x_path : array_like(float)
        An n x T matrix, where the t-th column represents x_t

    u_path : array_like(float)
        A k x T matrix, where the t-th column represents u_t

    w_path : array_like(float)
        A j x T matrix, where the t-th column represent w_t

    """
    # == Simplify notation == #
    A, B, C = self.A, self.B, self.C

    # == Preliminaries, finite horizon case == #
    if self.T:
        T = self.T if not ts_length else min(ts_length, self.T)
        self.P, self.d = self.Rf, 0

    # == Preliminaries, infinite horizon case == #
    else:
        T = ts_length if ts_length else 100
        self.stationary_values()

    # == Set up initial condition and arrays to store paths == #
    x0 = np.asarray(x0)

```

```

x0 = x0.reshape(self.n, 1) # Make sure x0 is a column vector
x_path = np.empty((self.n, T+1))
u_path = np.empty((self.k, T))
w_path = dot(C, np.random.randn(self.j, T+1))

# == Compute and record the sequence of policies == #
policies = []
for t in range(T):
    if self.T: # Finite horizon case
        self.update_values()
    policies.append(self.F)

# == Use policy sequence to generate states and controls == #
F = policies.pop()
x_path[:, 0] = x0.flatten()
u_path[:, 0] = - dot(F, x0).flatten()
for t in range(1, T):
    F = policies.pop()
    Ax, Bu = dot(A, x_path[:, t-1]), dot(B, u_path[:, t-1])
    x_path[:, t] = Ax + Bu + w_path[:, t]
    u_path[:, t] = - dot(F, x_path[:, t])
Ax, Bu = dot(A, x_path[:, T-1]), dot(B, u_path[:, T-1])
x_path[:, T] = Ax + Bu + w_path[:, T]

return x_path, u_path, w_path

```

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 (2.109), (2.110) and (2.111)
  - `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` from the [applications repository](#), 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: Nonstationary Income** 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

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (c_t - \bar{c})^2 + \beta^T q u_T^2 \right\} \quad (2.121)$$

subject to  $a_{t+1} = (1+r)a_t - c_t + y_t$ ,  $t \geq 0$

For income we now take  $y_t = p(t) + \sigma w_{t+1}$  where  $p(t) := m_0 + m_1 t + m_2 t^2$

(In the next section we employ some tricks to implement a more sophisticated model)

The coefficients  $m_0, m_1, m_2$  are chosen such that  $p(0) = 0$ ,  $p(T/2) = \mu$ , and  $p(T) = 0$

You can confirm that the specification  $m_0 = 0, m_1 = T\mu/(T/2)^2, m_2 = -\mu/(T/2)^2$  satisfies these constraints

To put this into an LQ setting, consider the budget constraint, which becomes

$$a_{t+1} = (1+r)a_t - u_t - \bar{c} + m_1 t + m_2 t^2 + \sigma w_{t+1} \quad (2.122)$$

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

$$x_t := \begin{pmatrix} a_t \\ 1 \\ t \\ t^2 \end{pmatrix}, \quad A := \begin{pmatrix} 1+r & -\bar{c} & m_1 & m_2 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad (2.123)$$

If you expand the expression  $x_{t+1} = Ax_t + Bu_t + Cw_{t+1}$  using this specification, you will find that assets follow (2.122) as desired, and that the other state variables also update appropriately

To implement preference specification (2.121) we take

$$Q := 1, \quad R := \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad R_f := \begin{pmatrix} q & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad (2.124)$$

The next figure shows a simulation of consumption and assets computed using the `compute_sequence` method of `lqcontrol.py` with initial assets set to zero

Once again, smooth consumption is a dominant feature of the sample paths

The asset path exhibits dynamics consistent with standard life cycle theory

Exercise 1 gives the full set of parameters used here and asks you to replicate the figure

**Application 2: A Permanent Income Model with Retirement** In the *previous application*, we generated income dynamics with an inverted U shape using polynomials, and placed them in an LQ framework

It is arguably the case that this income process still contains unrealistic features

A more common earning profile is where

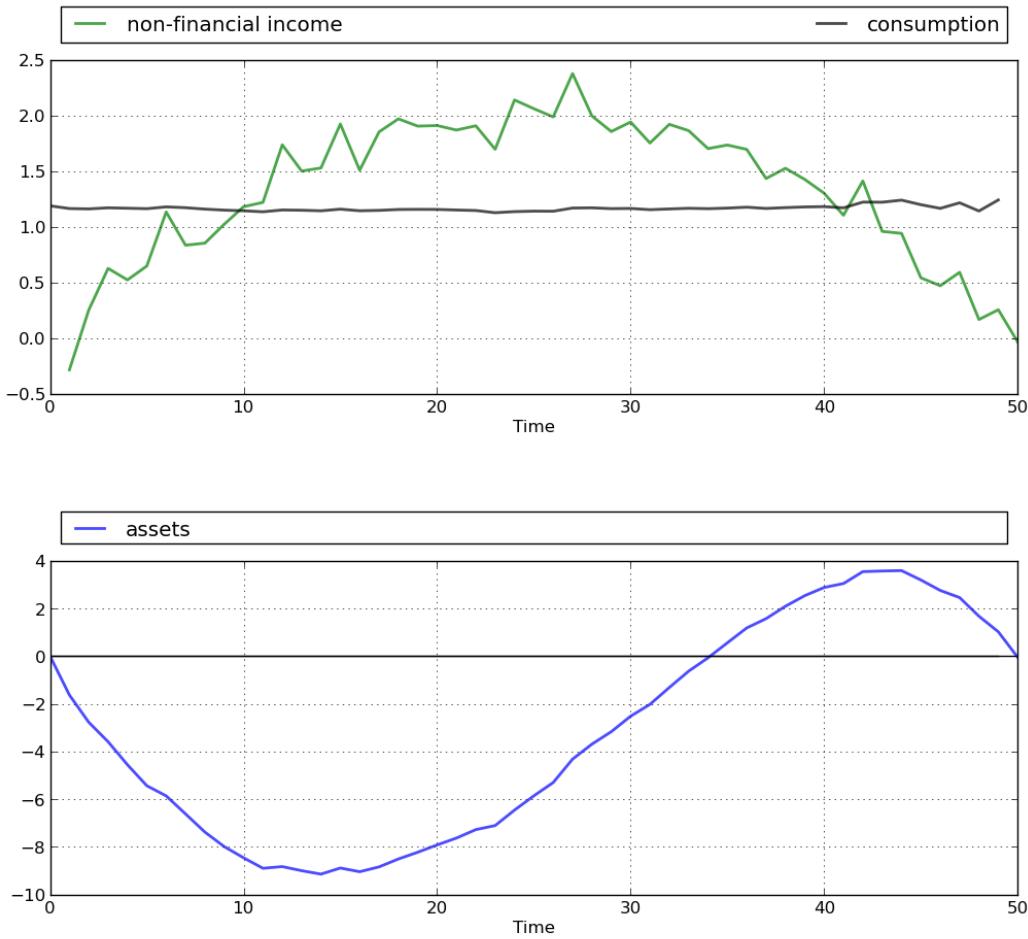
1. income grows over working life, fluctuating around an increasing trend, with growth flattening off in later years
2. retirement follows, with lower but relatively stable (non-financial) income

Letting  $K$  be the retirement date, we can express these income dynamics by

$$y_t = \begin{cases} p(t) + \sigma w_{t+1} & \text{if } t \leq K \\ s & \text{otherwise} \end{cases} \quad (2.125)$$

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 (2.113)

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 (2.125) 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

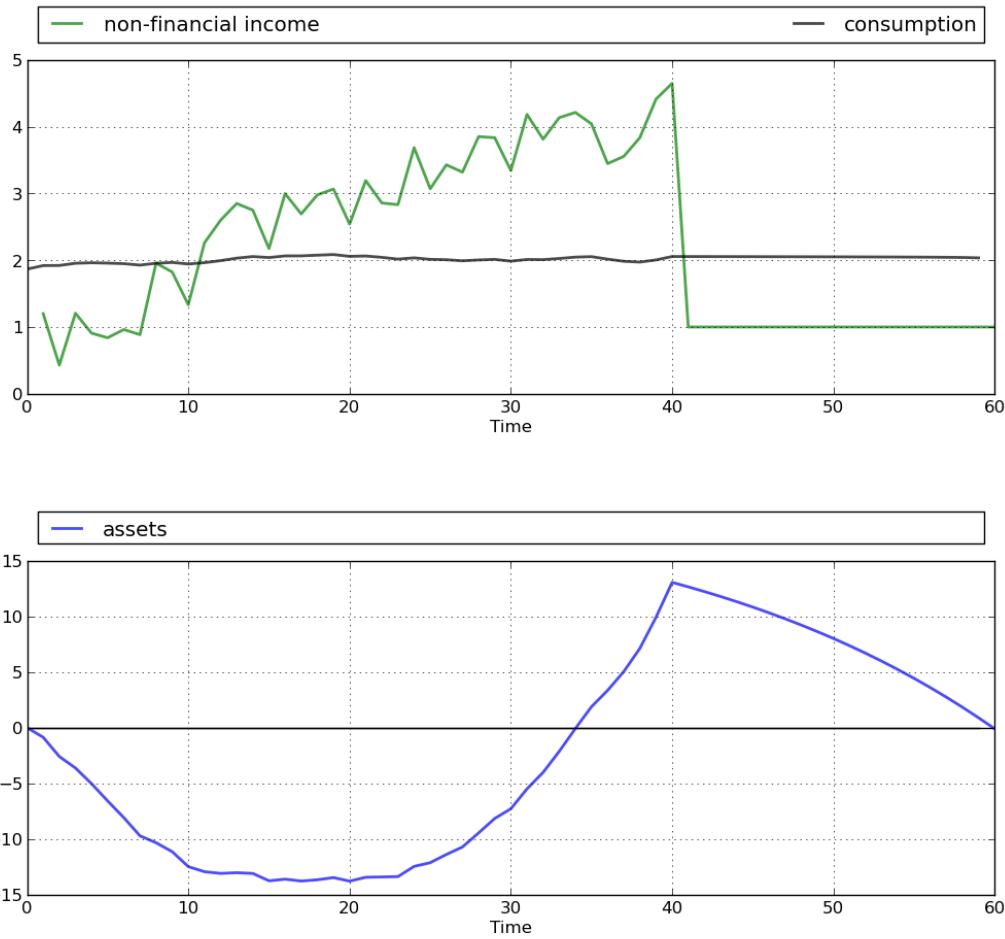
**Application 3: Monopoly with Adjustment Costs** Consider a monopolist facing stochastic inverse demand function

$$p_t = a_0 - a_1 q_t + d_t$$

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

$$\mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t \pi_t \right\} \quad \text{where} \quad \pi_t := p_t q_t - c q_t - \gamma (q_{t+1} - q_t)^2 \quad (2.126)$$

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  $\gamma$  is that

- if  $\gamma$  is close to zero, then  $q_t$  will track the time path of  $\bar{q}_t$  relatively closely
- if  $\gamma$  is larger, then  $q_t$  will be smoother than  $\bar{q}_t$ , as the monopolist seeks to avoid adjustment costs

This intuition turns out to be correct

The following figures show simulations produced by solving the corresponding LQ problem

The only difference in parameters across the figures is the size of  $\gamma$

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  $u_t = q_{t+1} - q_t$

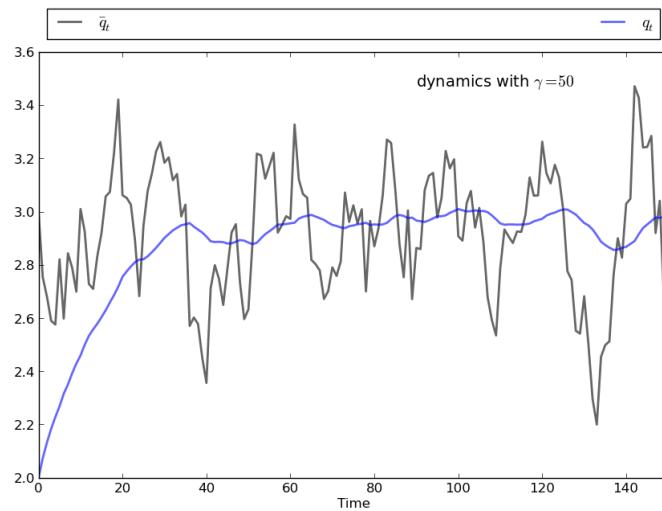
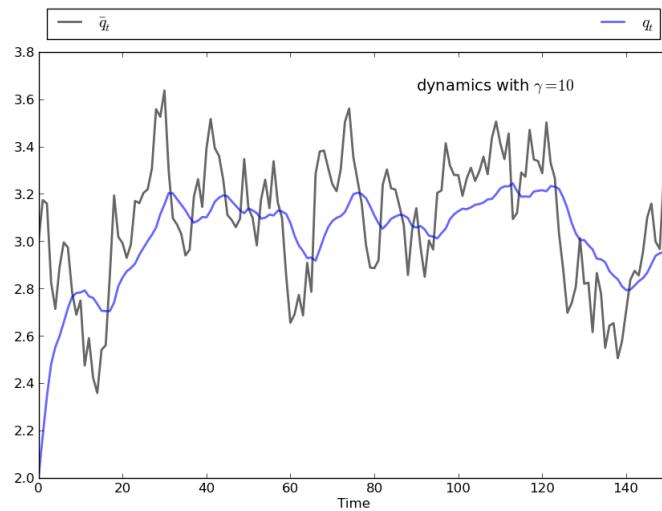
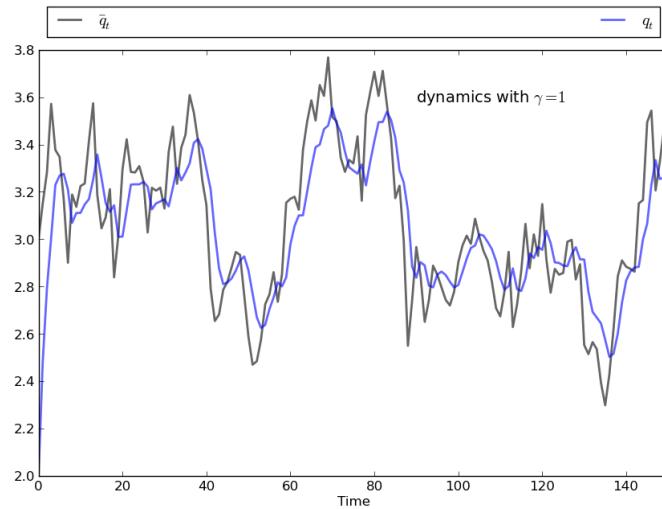
We also manipulated the profit function slightly

In (2.126), current profits are  $\pi_t := p_t q_t - c q_t - \gamma (q_{t+1} - q_t)^2$

Let's now replace  $\pi_t$  in (2.126) 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 (2.126), 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

$$\min \mathbb{E} \sum_{t=0}^{\infty} \beta^t \{ a_1(q_t - \bar{q}_t)^2 + \gamma u_t^2 \} \quad (2.127)$$

It's now relatively straightforward to find  $R$  and  $Q$  such that (2.127) can be written as (2.117)

Furthermore, the matrices  $A$ ,  $B$  and  $C$  from (2.98) 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 (2.123)

- 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 (2.124)

For `lq_working`, preferences are the same, except that  $R_f$  should be replaced by the final value function that emerges from iterating `lq_retired` back to the start of retirement

With some careful footwork, the simulation can be generated by patching together the simulations from these two separate models

**Exercise 3** Reproduce the figures from the monopolist application *given above*

For parameters, use  $a_0 = 5$ ,  $a_1 = 0.5$ ,  $\sigma = 0.15$ ,  $\rho = 0.9$ ,  $\beta = 0.95$  and  $c = 2$ , while  $\gamma$  varies between 1 and 50 (see figures)

## Solutions

[Solution notebook](#)

# Discrete Dynamic Programming

## Overview

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

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

We call such problems discrete dynamic programs, or simply 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** In this lecture we describe

- the theory of dynamic programming in a discrete setting
- a set of routines for solving discrete DPs from the [QuantEcon](#) code library
- examples and applications

The code discussed here 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

This lecture also draws extensively on documentation and examples written by [Daisuke Oyama](#)

**How to Read this Lecture** This lecture provides a detailed treatment of discrete dynamic programming

If you

- already know discrete dynamic programming, or
- want to move quickly to problem solving

then we suggest you *review the notation* and then skip immediately to *the first example*

Alternatively, if you find the treatment of dynamic programming too theoretical, you might want to review some of the introductory lectures on dynamic programming, such as

- The [shortest path lecture](#)
- The [lake model lecture](#)
- The [optimal growth lecture](#)

**References** For background reading 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t r(s_t, a_t) \quad (2.128)$$

where

- $s_t$  is the state variable
- $a_t$  is the action

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

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

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

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

Examples:

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

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

- A stationary Markov policy is a map  $\sigma$  from states to actions, with  $a_t = \sigma(s_t)$  indicating that  $a_t$  is the action to be taken in state  $s_t$
- 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 (2.128)

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*  $S = \{0, \dots, n - 1\}$
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*

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

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

A *policy function*, or simply *policy*, is a function  $\sigma: S \rightarrow A$

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

Denote the set of all feasible policies by  $\Sigma$

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

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

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

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

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

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

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

$$(Q_\sigma^t r_\sigma)(s) = \mathbb{E}[r(s_t, \sigma(s_t)) \mid s_0 = s] \quad \text{when } \{s_t\} \sim Q_\sigma \quad (2.129)$$

Comments

- $\{s_t\} \sim Q_\sigma$  means that the state is generated by stochastic matrix  $Q_\sigma$
- See *this discussion* on computing expectations of Markov chains for an explanation of the expression in (2.129)

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 the same thing (up to an isometric isomorphism!)

**Value and Optimality** Let  $v_\sigma(s)$  denote the discounted sum of expected reward flows from policy  $\sigma$  when the initial state is  $s$

To calculate this quantity we pass the expectation through the sum in (2.128) and use (2.129) to get

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

This function is called the *policy value function* for the policy  $\sigma$

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

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

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

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

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

$$\sigma(s) \in \arg \max_{a \in A(s)} \left\{ r(s, a) + \beta \sum_{s' \in S} w(s') Q(s, a, s') \right\} \quad (s \in S)$$

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

**Two Operators** It is useful to define the following operators:

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

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

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

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

This can be written more succinctly in operator notation as

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

The two operators are both monotone

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

They are also contraction mappings with modulus  $\beta$

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

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

For proofs of these results and those in the next section, see, for example, [EDTC](#), chapter 10

**The Bellman Equation and the Principle of Optimality** The main principle of the theory of dynamic programming is that

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

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

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

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

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

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

### Solving Discrete DPs

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

Code for solving discrete DPs is available in the `DiscreteDP` class from `QuantEcon.py`

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

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

Let's briefly review these algorithms and their implementation

**Value Function Iteration** Perhaps the most familiar method for solving all manner of dynamic programs is value function iteration

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

Hence, iterative application of  $T$  to any initial function  $v^0: S \rightarrow \mathbb{R}$  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  $v_\sigma$  of a policy  $\sigma$  by solving the linear equation  $v = T_\sigma v$
2. A policy improvement step that computes a  $v_\sigma$ -greedy policy

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

- See theorem 10.2.6 of `EDTC` for a proof

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

**Modified Policy Function Iteration** Modified policy iteration replaces the policy evaluation step in policy iteration with "partial policy evaluation"

The latter computes an approximation to the value of a policy  $\sigma$  by iterating  $T_\sigma$  for a specified number of times

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

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

### Example: A Growth Model

Let's consider a simple consumption-saving model

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

The household starts each period with current stock  $s$

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

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

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

The next period stock is therefore

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

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

**Discrete DP Representation** We want to represent this model in the format of a discrete dynamic program

To this end, we take

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

$$Q(s, a, s') := \begin{cases} \frac{1}{B+1} & \text{if } a \leq s' \leq a + B \\ 0 & \text{otherwise} \end{cases} \quad (2.130)$$

**Defining a DiscreteDP Instance** This information will be used to create an instance of *DiscreteDP* by passing the following information

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

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

For  $Q$  we follow the rule in (2.130)

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 the QuantEcon.applications repository

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(object):

    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

```
In [1]: run finite_dp_og_example.py

In [2]: 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

```
In [3]: import quantecon as qe

In [4]: 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

```
In [5]: results = ddp.solve(method='policy_iteration')
```

Let's see what we've got here

```
In [6]: dir(results)
Out[6]: ['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

```
In [7]: results.v
Out[7]:
array([ 19.01740222,  20.01740222,  20.43161578,  20.74945302,
       21.04078099,  21.30873018,  21.54479816,  21.76928181,
       21.98270358,  22.18824323,  22.3845048 ,  22.57807736,
       22.76109127,  22.94376708,  23.11533996,  23.27761762])

In [8]: results.sigma
Out[8]: array([0, 0, 0, 0, 1, 1, 1, 2, 2, 3, 3, 4, 5, 5, 5])
```

Since we've used policy iteration, these results will be exact unless we hit the iteration bound max\_iter

Let's make sure this didn't happen

```
In [9]: results.max_iter
Out[9]: 250
```

```
In [10]: results.num_iter
Out[10]: 3
```

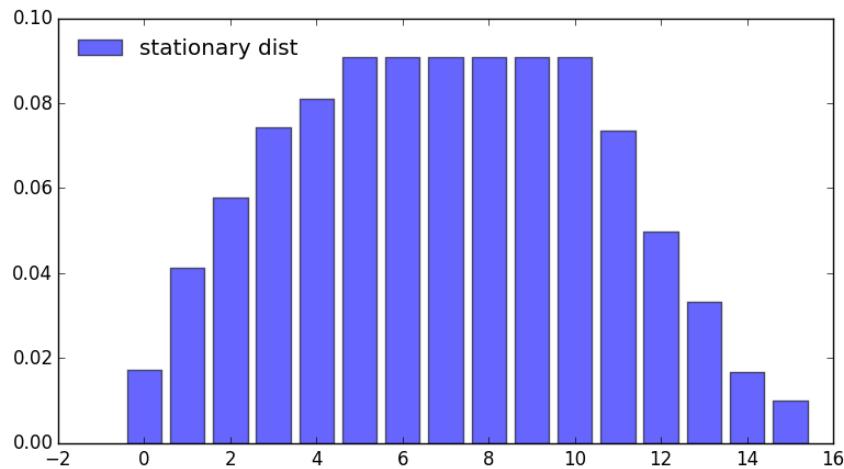
Another interesting object is `results.mc`, which is the controlled chain defined by  $Q_{\sigma^*}$ , where  $\sigma^*$  is the optimal policy

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

Since this object is an instance of the `MarkovChain` class from `QuantEcon` (see [this lecture](#) for more discussion), we can immediately simulate it, compute its stationary distribution and so on

```
In [11]: results.mc.stationary_distributions
Out[11]:
array([[ 0.01732187,  0.04121063,  0.05773956,  0.07426848,  0.08095823,
        0.09090909,  0.09090909,  0.09090909,  0.09090909,  0.09090909,
        0.09090909,  0.07358722,  0.04969846,  0.03316953,  0.01664061,
        0.00995086]])
```

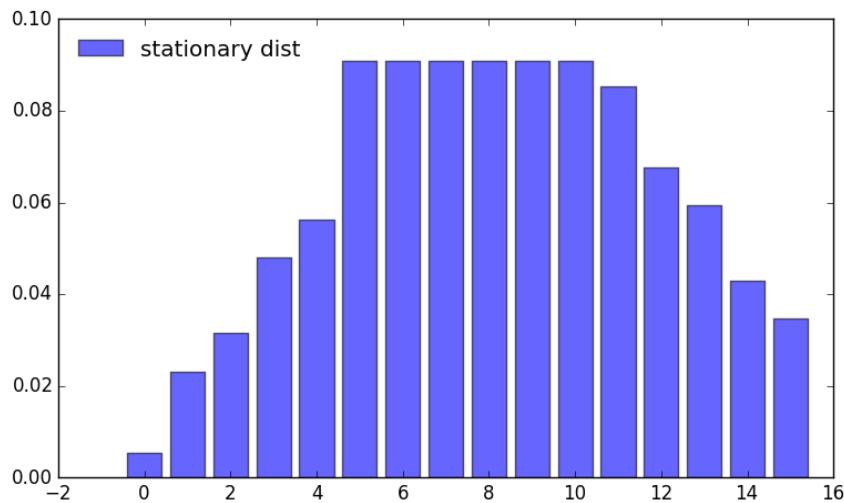
Here's the same information in a bar graph



What happens if the agent is more patient?

```
In [18]: ddp = qe.markov.DiscreteDP(g.R, g.Q, 0.99) # Increase beta to 0.99
In [19]: results = ddp.solve(method='policy_iteration')
In [20]: results.mc.stationary_distributions
Out[20]:
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  $L \times n$  transition probability array

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

```
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.append([0] * m)
        R.append(0)

for s in range(n):
    for a in range(min(M, s) + 1):
        if a == 0:
            Q[s][a] = b
            R[s] += b * u(s)
        else:
            Q[s][a] = (1 - beta) * b
            R[s] += (1 - beta) * b * u(s)
            for j in range(1, a):
                Q[s][a] += beta * b
                R[s] += beta * b * u(s)
            Q[s][a] += beta
            R[s] += beta * u(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 again using the `DiscreteDP` class described above

### Solutions

[Solution notebook](#)

### Appendix: Algorithms

This appendix covers the details of the solution algorithms implemented in the `DiscreteDP` class

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

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

**Value Iteration** The `DiscreteDP.value_iteration` method implements value function iteration as follows

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

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

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

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

**Policy Iteration** The implementation in the method `DiscreteDP.policy_iteration` runs as follows

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

The policy iteration algorithm terminates in a finite number of iterations

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

**Modified Policy Iteration** The implementation in the `DiscreteDP.modified_policy_iteration` method runs as follows:

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

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

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

See also the documentation for `DiscreteDP`

## 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

$$p = a_0 - a_1 Y \quad (2.131)$$

where

- $a_i > 0$  for  $i = 0, 1$
- $Y = ny$  is the market-wide level of output

Each firm has total cost function

$$c(y) = c_1 y + 0.5 c_2 y^2, \quad c_i > 0 \text{ for } i = 1, 2$$

The profits of a representative firm are  $py - c(y)$

Using (2.131), we can express the problem of the representative firm as

$$\max_y [(a_0 - a_1 Y)y - c_1 y - 0.5 c_2 y^2] \quad (2.132)$$

In posing problem (2.132), we want the firm to be a *price taker*

We do that by regarding  $p$  and therefore  $Y$  as exogenous to the firm

The essence of the Big  $Y$ , little  $y$  trick is *not* to set  $Y = ny$  before taking the first-order condition with respect to  $y$  in problem (2.132)

This assures that the firm is a price taker

The first order condition for problem (2.132) is

$$a_0 - a_1 Y - c_1 - c_2 y = 0 \quad (2.133)$$

At this point, *but not before*, we substitute  $Y = ny$  into (2.133) to obtain the following linear equation

$$a_0 - c_1 - (a_1 + n^{-1} c_2) Y = 0 \quad (2.134)$$

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 (2.131)

**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

$$p_t = a_0 - a_1 Y_t \quad (2.135)$$

where

- $a_i > 0$  for  $i = 0, 1$
- $Y_t = ny_t$  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

$$\sum_{t=0}^{\infty} \beta^t r_t \quad (2.136)$$

where

$$r_t := p_t y_t - \frac{\gamma(y_{t+1} - y_t)^2}{2}, \quad y_0 \text{ given} \quad (2.137)$$

Regarding the parameters,

- $\beta \in (0, 1)$  is a discount factor
- $\gamma > 0$  measures the cost of adjusting the rate of output

Regarding timing, the firm observes  $p_t$  and  $y_t$  when it chooses  $y_{t+1}$  at time  $t$

To state the firm's optimization problem completely requires that we specify dynamics for all state variables

This includes ones that the firm cares about but does not control like  $p_t$

We turn to this problem now

**Prices and Aggregate Output** In view of (2.135), 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

$$Y_{t+1} = H(Y_t) \quad (2.138)$$

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 (2.138) 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

$$v(y, Y) = \max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \quad (2.139)$$

Let's denote the firm's optimal policy function by  $h$ , so that

$$y_{t+1} = h(y_t, Y_t) \quad (2.140)$$

where

$$h(y, Y) := \arg \max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \quad (2.141)$$

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

$$-\gamma(y' - y) + \beta v_y(y', H(Y)) = 0 \quad (2.142)$$

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 (2.139), giving

$$v_y(y, Y) = a_0 - a_1 Y + \gamma(y' - y)$$

Substituting this equation into (2.142) gives the *Euler equation*

$$-\gamma(y_{t+1} - y_t) + \beta[a_0 - a_1 Y_{t+1} + \gamma(y_{t+2} - y_{t+1})] = 0 \quad (2.143)$$

The firm optimally sets an output path that satisfies (2.143), taking (2.138) as given, and subject to

- the initial conditions for  $(y_0, Y_0)$
- the terminal condition  $\lim_{t \rightarrow \infty} \beta^t y_t v_y(y_t, Y_t) = 0$

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 (2.143) subject to the given initial condition  $y_0$  and the transversality condition

Note that solving the Bellman equation (2.139) for  $v$  and then  $h$  in (2.141) yields a decision rule that automatically imposes both the Euler equation (2.143) 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

$$Y_{t+1} = nh(Y_t/n, Y_t) \quad (2.144)$$

Thus, when firms believe that the law of motion for market-wide output is (2.138), their optimizing behavior makes the actual law of motion be (2.144)

**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 (2.138) and (2.144)

**Fixed point characterization** As we've seen, the firm's optimum problem induces a mapping  $\Phi$  from a perceived law of motion  $H$  for market-wide output to an actual law of motion  $\Phi(H)$

The mapping  $\Phi$  is the composition of two operations, taking a perceived law of motion into a decision rule via (2.139)–(2.141), and a decision rule into an actual law via (2.144)

The  $H$  component of a rational expectations equilibrium is a fixed point of  $\Phi$

### Computation of an Equilibrium

Now let's consider the problem of computing the rational expectations equilibrium

**Misbehavior of  $\Phi$**  Readers accustomed to dynamic programming arguments might try to address this problem by choosing some guess  $H_0$  for the aggregate law of motion and then iterating with  $\Phi$

Unfortunately, the mapping  $\Phi$  is not a contraction

In particular, there is no guarantee that direct iterations on  $\Phi$  converge<sup>1</sup>

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  $t$

$$s(Y_t, Y_{t+1}) := \int_0^{Y_t} (a_0 - a_1 x) dx - \frac{\gamma(Y_{t+1} - Y_t)^2}{2} \quad (2.145)$$

The first term is the area under the demand curve, while the second measures the social costs of changing output

---

<sup>1</sup> A literature that studies whether models populated with agents who learn can converge to rational expectations equilibria features iterations on a modification of the mapping  $\Phi$  that can be approximated as  $\gamma\Phi + (1 - \gamma)I$ . Here  $I$  is the identity operator and  $\gamma \in (0, 1)$  is a *relaxation parameter*. See [MS89] and [EH01] for statements and applications of this approach to establish conditions under which collections of adaptive agents who use least squares learning converge to a rational expectations equilibrium.

The *planning problem* is to choose a production plan  $\{Y_t\}$  to maximize

$$\sum_{t=0}^{\infty} \beta^t s(Y_t, Y_{t+1})$$

subject to an initial condition for  $Y_0$

**Solution of the Planning Problem** Evaluating the integral in (2.145) 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

$$V(Y) = \max_{Y'} \left\{ a_0 Y - \frac{a_1}{2} Y^2 - \frac{\gamma(Y' - Y)^2}{2} + \beta V(Y') \right\} \quad (2.146)$$

The associated first order condition is

$$-\gamma(Y' - Y) + \beta V'(Y') = 0 \quad (2.147)$$

Applying the same Benveniste-Scheinkman formula gives

$$V'(Y) = a_0 - a_1 Y + \gamma(Y' - Y)$$

Substituting this into equation (2.147) and rearranging leads to the Euler equation

$$\beta a_0 + \gamma Y_t - [\beta a_1 + \gamma(1 + \beta)] Y_{t+1} + \gamma \beta Y_{t+2} = 0 \quad (2.148)$$

**The Key Insight** Return to equation (2.143) 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 (2.148) and (2.143) 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  $Y_t = ny_t$  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 (2.146)

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

$$Y_{t+1} = \kappa_0 + \kappa_1 Y_t \quad (2.149)$$

for some parameter pair  $\kappa_0, \kappa_1$

Now that we know the aggregate law of motion is linear, we can see from the firm's Bellman equation (2.139) 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

$$y_{t+1} = h_0 + h_1 y_t + h_2 Y_t \quad (2.150)$$

Hence a rational expectations equilibrium will be defined by the parameters  $(\kappa_0, \kappa_1, h_0, h_1, h_2)$  in (2.149)–(2.150)

### Exercises

**Exercise 1** Consider the firm problem *described above*

Let the firm's belief function  $H$  be as given in (2.149)

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 (2.150) and give the values for each  $h_j$

If there were  $n$  identical competitive firms all behaving according to (2.150), what would (2.150) imply for the *actual* law of motion (2.138) for market supply

**Exercise 2** Consider the following  $\kappa_0, \kappa_1$  pairs as candidates for the aggregate law of motion component of a rational expectations equilibrium (see (2.149))

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 (2.135) 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

[Solution notebook](#)

# 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

$$p = a_0 - a_1(q_1 + q_2) \quad (2.151)$$

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 (2.151) and what follows,

- the time subscript is suppressed when possible to simplify notation
- $\hat{x}$  denotes a next period value of variable  $x$

Each firm recognizes that its output affects total output and therefore the market price

The one-period payoff function of firm  $i$  is price times quantity minus adjustment costs:

$$\pi_i = pq_i - \gamma(\hat{q}_i - q_i)^2, \quad \gamma > 0, \quad (2.152)$$

Substituting the inverse demand curve (2.151) into (2.152) lets us express the one-period payoff as

$$\pi_i(q_i, q_{-i}, \hat{q}_i) = a_0 q_i - a_1 q_i^2 - a_1 q_i q_{-i} - \gamma(\hat{q}_i - q_i)^2, \quad (2.153)$$

where  $q_{-i}$  denotes the output of the firm other than  $i$

The objective of the firm is to maximize  $\sum_{t=0}^{\infty} \beta^t \pi_{it}$

Firm  $i$  chooses a decision rule that sets next period quantity  $\hat{q}_i$  as a function  $f_i$  of the current state  $(q_i, q_{-i})$

An essential aspect of a Markov perfect equilibrium is that each firm takes the decision rule of the other firm as known and given

Given  $f_{-i}$ , the Bellman equation of firm  $i$  is

$$v_i(q_i, q_{-i}) = \max_{\hat{q}_i} \{ \pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i(\hat{q}_i, f_{-i}(q_{-i}, q_i)) \} \quad (2.154)$$

**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 (2.154)
- The maximizer on the right side of (2.154) 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

$$v_i^{j+1}(q_i, q_{-i}) = \max_{\hat{q}_i} \{ \pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i^j(\hat{q}_i, f_{-i}(q_{-i}, q_i)) \} \quad (2.155)$$

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

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2x_t' W_i u_{it} + 2u_{-it}' M_i u_{it}\} \quad (2.156)$$

while the state evolves according to

$$x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t} \quad (2.157)$$

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  $t = t_0, \dots, t_1 - 1$  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 (2.156) and (2.157), then player 1's problem becomes minimization of

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' \Pi_{1t} x_t + u_{1t}' Q_1 u_{1t} + 2u_{1t}' \Gamma_{1t} x_t\} \quad (2.158)$$

subject to

$$x_{t+1} = \Lambda_{1t} x_t + B_1 u_{1t}, \quad (2.159)$$

where

- $\Lambda_{it} := A - B_{-it} F_{-it}$
- $\Pi_{it} := R_i + F_{-it}' S_i F_{-it}$
- $\Gamma_{it} := W_i' - M_i' F_{-it}$

This is an LQ dynamic programming problem that can be solved by working backwards

The policy rule that solves this problem is

$$F_{1t} = (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) \quad (2.160)$$

where  $P_{1t}$  is the solution of the matrix Riccati difference equation

$$P_{1t} = \Pi_{1t} - (\beta B'_1 P_{1t+1} \Lambda_{1t} + \Gamma_{1t})' (Q_1 + \beta B'_1 P_{1t+1} B_1)^{-1} (\beta B'_1 P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) + \beta \Lambda'_{1t} P_{1t+1} \Lambda_{1t} \quad (2.161)$$

Similarly, the policy that solves player 2's problem is

$$F_{2t} = (Q_2 + \beta B'_2 P_{2t+1} B_2)^{-1} (\beta B'_2 P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) \quad (2.162)$$

where  $P_{2t}$  solves

$$P_{2t} = \Pi_{2t} - (\beta B'_2 P_{2t+1} \Lambda_{2t} + \Gamma_{2t})' (Q_2 + \beta B'_2 P_{2t+1} B_2)^{-1} (\beta B'_2 P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) + \beta \Lambda'_{2t} P_{2t+1} \Lambda_{2t} \quad (2.163)$$

Here in all cases  $t = t_0, \dots, t_1 - 1$  and the initial conditions are  $P_{it_1} = 0$

The solution procedure is to use equations (2.160), (2.161), (2.162), and (2.163), 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 (2.160) contain  $F_{2t}$
- some terms on the right hand side of (2.162) contain  $F_{1t}$

we need to solve these  $k_1 + k_2$  equations simultaneously

A key insight is that the equations (2.160) and (2.162) 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 (2.161) and (2.163)

**Infinite horizon** We often want to compute the solutions of such games for infinite horizons, in the hope that the decision rules  $F_{it}$  settle down to be time invariant as  $t_1 \rightarrow +\infty$

In practice, we usually fix  $t_1$  and compute the equilibrium of an infinite horizon game by driving  $t_0 \rightarrow -\infty$

This is the approach we adopt in the next section

**Implementation** Below we display a function called *nnash* that computes a Markov perfect equilibrium of the infinite horizon linear quadratic dynamic game in the manner described above

```
from __future__ import division, print_function
import numpy as np
from numpy import dot, eye
from scipy.linalg import solve

def nnash(A, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2,
          beta=1.0, tol=1e-8, max_iter=1000):
    """
    Compute the limit of a Nash linear quadratic dynamic game. In this
    problem, player i minimizes
    """
    # ... (rest of the code)
```

```

.. math::
\sum_{t=0}^{\infty}
\left\{
x_t' r_i x_t + 2 x_t' w_i
u_{it}' q_i u_{it} + u_{jt}' s_i u_{jt} + 2 u_{jt}' m_i u_{it}
\right\}

```

subject to the law of motion

```

.. math::
x_{t+1} = A x_t + b_1 u_{1t} + b_2 u_{2t}

```

and a perceived control law :math:`u\_j(t) = -f\_j x\_t` for the other player.

The solution computed in this routine is the :math:`f\_i` and :math:`p\_i` of the associated double optimal linear regulator problem.

#### Parameters

-----

*A* : scalar(float) or array\_like(float)  
 Corresponds to the above equation, should be of size (n, n)

*B1* : scalar(float) or array\_like(float)  
 As above, size (n, k\_1)

*B2* : scalar(float) or array\_like(float)  
 As above, size (n, k\_2)

*R1* : scalar(float) or array\_like(float)  
 As above, size (n, n)

*R2* : scalar(float) or array\_like(float)  
 As above, size (n, n)

*Q1* : scalar(float) or array\_like(float)  
 As above, size (k\_1, k\_1)

*Q2* : scalar(float) or array\_like(float)  
 As above, size (k\_2, k\_2)

*S1* : scalar(float) or array\_like(float)  
 As above, size (k\_1, k\_1)

*S2* : scalar(float) or array\_like(float)  
 As above, size (k\_2, k\_2)

*W1* : scalar(float) or array\_like(float)  
 As above, size (n, k\_1)

*W2* : scalar(float) or array\_like(float)  
 As above, size (n, k\_2)

*M1* : scalar(float) or array\_like(float)  
 As above, size (k\_2, k\_1)

*M2* : scalar(float) or array\_like(float)  
 As above, size (k\_1, k\_2)

*beta* : scalar(float), optional(default=1.0)  
 Discount rate

*tol* : scalar(float), optional(default=1e-8)  
 This is the tolerance level for convergence

*max\_iter* : scalar(int), optional(default=1000)

```

This is the maximum number of iterations allowed

>Returns
-----
F1 : array_like, dtype=float, shape=(k_1, n)
    Feedback law for agent 1
F2 : array_like, dtype=float, shape=(k_2, n)
    Feedback law for agent 2
P1 : array_like, dtype=float, shape=(n, n)
    The steady-state solution to the associated discrete matrix
    Riccati equation for agent 1
P2 : array_like, dtype=float, shape=(n, n)
    The steady-state solution to the associated discrete matrix
    Riccati equation for agent 2

"""
# == Upload parameters and make sure everything is an array == #
params = A, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2
params = map(np.asarray, params)
A, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2 = params

# == Multiply A, B1, B2 by sqrt(beta) to enforce discounting == #
A, B1, B2 = [np.sqrt(beta) * x for x in (A, B1, B2)]

n = A.shape[0]

if B1.ndim == 1:
    k_1 = 1
    B1 = np.reshape(B1, (n, 1))
else:
    k_1 = B1.shape[1]

if B2.ndim == 1:
    k_2 = 1
    B2 = np.reshape(B2, (n, 1))
else:
    k_2 = B2.shape[1]

v1 = eye(k_1)
v2 = eye(k_2)
P1 = np.zeros((n, n))
P2 = np.zeros((n, n))
F1 = np.random.randn(k_1, n)
F2 = np.random.randn(k_2, n)

for it in range(max_iter):
    # update
    F10 = F1
    F20 = F2

    G2 = solve(dot(B2.T, P2.dot(B2))+Q2, v2)
    G1 = solve(dot(B1.T, P1.dot(B1))+Q1, v1)
    H2 = dot(G2, B2.T.dot(P2))

```

```

H1 = dot(G1, B1.T.dot(P1))

# break up the computation of F1, F2
F1_left = v1 - dot(H1.dot(B2)+G1.dot(M1.T),
                     H2.dot(B1)+G2.dot(M2.T))
F1_right = H1.dot(A)+G1.dot(W1.T) - dot(H1.dot(B2)+G1.dot(M1.T),
                                         H2.dot(A)+G2.dot(W2.T))
F1 = solve(F1_left, F1_right)
F2 = H2.dot(A)+G2.dot(W2.T) - dot(H2.dot(B1)+G2.dot(M2.T), F1)

Lambda1 = A - B2.dot(F2)
Lambda2 = A - B1.dot(F1)
Pi1 = R1 + dot(F2.T, S1.dot(F2))
Pi2 = R2 + dot(F1.T, S2.dot(F1))

P1 = dot(Lambda1.T, P1.dot(Lambda1)) + Pi1 - \
     dot(dot(Lambda1.T, P1.dot(B1)) + W1 - F2.T.dot(M1), F1)
P2 = dot(Lambda2.T, P2.dot(Lambda2)) + Pi2 - \
     dot(dot(Lambda2.T, P2.dot(B2)) + W2 - F1.T.dot(M2), F2)

dd = np.max(np.abs(F10 - F1)) + np.max(np.abs(F20 - F2))

if dd < tol: # success!
    break

else:
    msg = 'No convergence: Iteration limit of {} reached in nnash'
    raise ValueError(msg.format(max_iter))

return F1, F2, P1, P2

```

## 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 (2.153)

The law of motion for the state  $x_t$  is  $x_{t+1} = Ax_t + B_1u_{1t} + B_2u_{2t}$  where

$$A := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B_1 := \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad B_2 := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The optimal decision rule of firm  $i$  will take the form  $u_{it} = -F_i x_t$ , inducing the following closed loop system for the evolution of  $x$  in the Markov perfect equilibrium:

$$x_{t+1} = (A - B_1 F_1 - B_2 F_2) x_t \quad (2.164)$$

**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.],
      [0.,  0.,  0.],
      [0.,  0.,  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

```

In [1]: 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 (2.158) and (2.159) to get firm 1's problem and solve it using  $LQ$

We hope that the resulting policy will agree with  $F1$  as computed above

```

In [2]: Lambda1 = A - np.dot(B2, F2)

In [3]: lq1 = qe.LQ(Q1, R1, Lambda1, B1, beta=beta)

In [4]: P1_ih, F1_ih, d = lq1.stationary_values()

In [5]: F1_ih
Out[5]: 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

```

In [6]: np.allclose(F1, F1_ih)
Out[6]: 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 (2.164)

The following program

- imports  $F1$  and  $F2$  from the previous program along with all parameters
- computes the evolution of  $x_t$  using (2.164)
- extracts and plots industry output  $q_t = q_{1t} + q_{2t}$  and price  $p_t = a_0 - a_1 q_t$

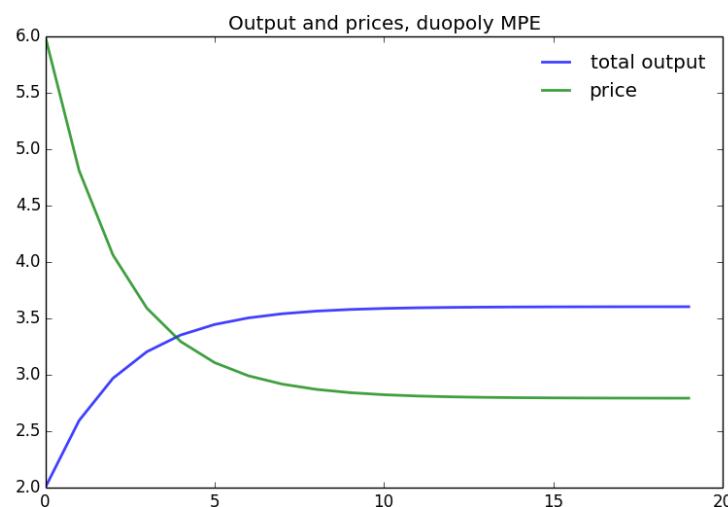
```
import matplotlib.pyplot as plt
from duopoly_mpe import *

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  $q_{10} = q_{20} = 1.0$

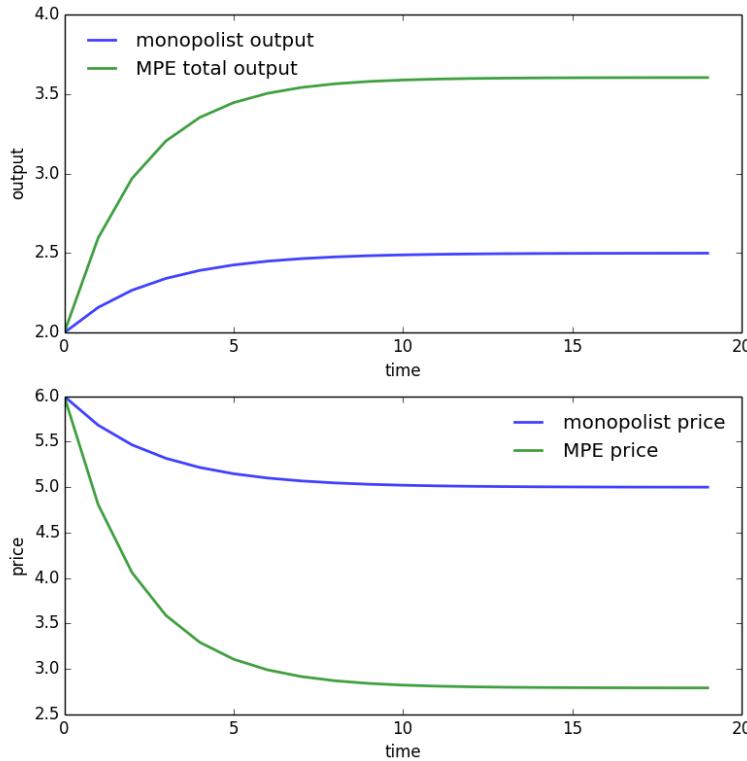
The resulting figure looks as follows



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 [Judd90]

Two firms set prices and quantities of two goods interrelated through their demand curves

Relevant variables are defined as follows:

- $I_{it}$  = inventories of firm  $i$  at beginning of  $t$
- $q_{it}$  = production of firm  $i$  during period  $t$
- $p_{it}$  = price charged by firm  $i$  during period  $t$
- $S_{it}$  = sales made by firm  $i$  during period  $t$
- $E_{it}$  = costs of production of firm  $i$  during period  $t$
- $C_{it}$  = costs of carrying inventories for firm  $i$  during  $t$

The firms' cost functions are

- $C_{it} = c_{i1} + c_{i2}I_{it} + 0.5c_{i3}I_{it}^2$
- $E_{it} = e_{i1} + e_{i2}q_{it} + 0.5e_{i3}q_{it}^2$  where  $e_{ij}, c_{ij}$  are positive scalars

Inventories obey the laws of motion

$$I_{i,t+1} = (1 - \delta)I_{it} + q_{it} - S_{it}$$

Demand is governed by the linear schedule

$$S_t = Dp_{it} + b$$

where

- $S_t = [S_{1t} \quad S_{2t}]'$
- $D$  is a  $2 \times 2$  negative definite matrix and
- $b$  is a vector of constants

Firm  $i$  maximizes the undiscounted sum

$$\lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=0}^T (p_{it}S_{it} - E_{it} - C_{it})$$

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

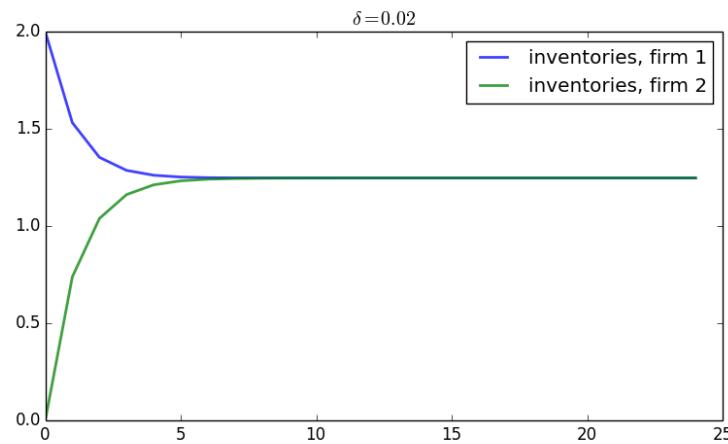
The exercise is to calculate these matrices and compute the following figures

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])

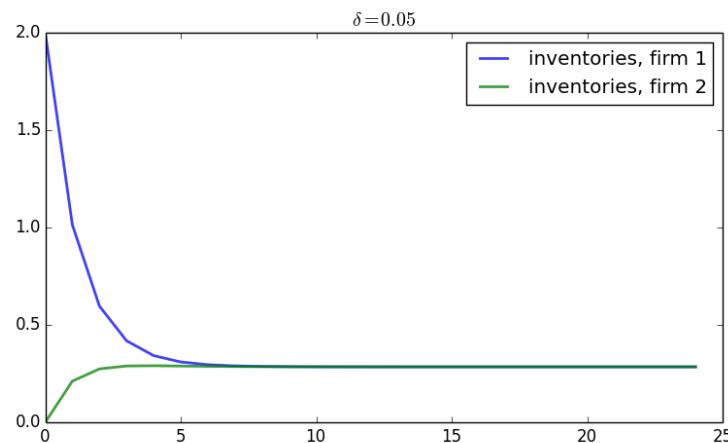
```



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

[Solution notebook](#)

## Markov Asset Pricing

## Contents

- *Markov Asset Pricing*
  - *Overview*
  - *Pricing Models*
  - *Classes of Assets*
  - *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 a stream of prospective payments

The spot price of an asset depends primarily on

- the anticipated dynamics for the stream of income accruing to the owners
- attitudes to risk and 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

We begin with some notation and then proceed to foundational 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  $d_t, d_{t+1}, \dots$
- A time- $t$  **ex-dividend** asset is a claim to the stream  $d_{t+1}, d_{t+2}, \dots$

**Risk Neutral Pricing** Let  $\beta = 1/(1 + \rho)$  be an intertemporal discount factor

In other words,  $\rho$  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

$$p_t = d_t + \beta \mathbb{E}_t[p_{t+1}] \quad (2.165)$$

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  $t$

In the present case this information set consists of observations of dividends up until time  $t$

For an ex-dividend asset, the basic risk-neutral asset pricing equation is

$$p_t = \beta \mathbb{E}_t[d_{t+1} + p_{t+1}] \quad (2.166)$$

**Pricing with Random Discount Factor** What happens if for some reason traders discount pay-outs differently depending on the state of the world?

Suppose that all agents evaluate payoffs according to a strictly concave period utility function  $u$

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

$$p_t = \mathbb{E}_t[m_{t+1}(d_{t+1} + p_{t+1})] \quad (2.167)$$

where  $m_{t+1}$  is a **stochastic discount factor**

Comparing (2.166) and (2.167), the difference is that the fixed discount factor  $\beta$  in (2.166) has been replaced by the stochastic discount factor  $m_{t+1}$

We give examples of how the stochastic discount factor has been modeled below

**Asset Pricing and Covariances** First a reminder. From the definition of a conditional covariance  $\text{cov}_t(x_{t+1}, y_{t+1})$  it follows that

$$\mathbb{E}_t(x_{t+1}y_{t+1}) = \text{cov}_t(x_{t+1}, y_{t+1}) + \mathbb{E}_t x_{t+1} \mathbb{E}_t y_{t+1} \quad (2.168)$$

If we apply this definition to the asset pricing equation (2.167) we obtain

$$p_t = \mathbb{E}_t m_{t+1} \mathbb{E}_t(d_{t+1} + p_{t+1}) + \text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1}) \quad (2.169)$$

It is useful to regard equation (2.169) as a generalization of equation (2.166)

- In equation (2.166), the stochastic discount factor  $m_{t+1} = \beta$ , a constant
- In equation (2.166), the covariance term  $\text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1})$  is zero because  $m_{t+1} = \beta$

Equation (2.169) 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](#)

For now let's study some of the implications of equation (2.167) for asset prices

**Simple Examples** What price dynamics result from these models?

The answer to this question depends on

1. the process we specify for dividends
2. the stochastic discount factor and how it is correlated with dividends

Let's look at some examples that illustrate this idea

**Example 1: Constant dividends, risk neutral pricing** The simplest case is a constant, non-random dividend stream  $d_t = d > 0$

Removing the expectation from (2.165) 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

$$\bar{p} := \frac{1}{1-\beta} d \quad (2.170)$$

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: Deterministic dividends, risk neutral pricing** Consider a growing, non-random dividend process  $d_t = \lambda^t d_0$  where  $0 < \lambda \beta < 1$

The cum-dividend price under risk neutral pricing is then

$$p_t = \frac{d_t}{1 - \beta \lambda} = \frac{\lambda^t d_0}{1 - \beta \lambda} \quad (2.171)$$

To obtain this,

1. set  $v_t = p_t/d_t$  in (2.165) and then  $v_t = v_{t+1} = v$  to solve for constant  $v$
2. recover the price as  $p_t = vd_t$

The ex-dividend price is  $p_t = (1 - \beta \lambda)^{-1} \beta \lambda d_t$

If, in this example, we take  $\lambda = 1 + g$  and let  $\rho := 1/\beta - 1$ , then the ex-dividend price becomes

$$p_t = \frac{1+g}{\rho-g} d_t$$

This is called the *Gordon formula*

**Example 3: Markov growth, risk neutral pricing** Next we consider a dividend process where the growth rate is Markovian

In particular,

$$d_{t+1} = \lambda_{t+1} d_t \quad \text{where} \quad \mathbb{P}\{\lambda_{t+1} = s_j \mid \lambda_t = s_i\} = P_{ij} := P[i, j]$$

This notation means that  $\{\lambda_t\}$  is an  $n$  state **Markov chain** with transition matrix  $P$  and state space  $s = \{s_1, \dots, s_n\}$

To obtain asset prices under risk neutrality, recall that in (2.171) the price dividend ratio  $p_t/d_t$  is constant and depends on  $\lambda$

This encourages us to guess that, in the current case,  $p_t/d_t$  is constant given  $\lambda_t$

That is  $p_t = v(\lambda_t)d_t$  for some unknown function  $v$  on the state space

To simplify notation, let  $v_i := v(s_i)$

For a cum-dividend stock we find that  $v_i = 1 + \beta \sum_{j=1}^n P_{ij} s_j v_j$

Letting  $\mathbf{1}$  be an  $n \times 1$  vector of ones and  $\tilde{P}_{ij} = P_{ij} s_j$ , we can express this in matrix notation as

$$v = (I - \beta \tilde{P})^{-1} \mathbf{1}$$

Here we are assuming invertibility, which *requires that* the growth rate of the Markov chain is not too large relative to  $\beta$

(In particular, that the eigenvalues of  $\tilde{P}$  be strictly less than  $\beta^{-1}$  in modulus)

Similar reasoning yields the ex-dividend price-dividend ratio  $w$ , which satisfies

$$w = \beta(I - \beta \tilde{P})^{-1} P s'$$

**Example 4: Deterministic dividends** Consider a non random dividend stream  $d_t = \lambda^t d$  for some  $\lambda > 0$

Furthermore, suppose a (nonstochastic) discount factor  $m_{t+1} = \beta \lambda^{-\gamma}$  for  $\gamma \geq 1$

Our formula for pricing a cum-dividend claim to the stream  $d_t = \lambda^t d$  becomes

$$p_t = d_t + \beta \lambda^{-\gamma} p_{t+1}$$

Guessing again that the price obeys  $p_t = v d_t$  where  $v$  is a constant price-dividend ratio, we have  $v d_t = d_t + \beta \lambda^{-\gamma} v d_{t+1}$ , or

$$v = \frac{1}{1 - \beta \lambda^{1-\gamma}}$$

If  $\gamma = 1$ , then the preceding formula for the price-dividend ratio becomes  $v = 1/(1 - \beta)$

Here the price-dividend ratio is constant and independent of the dividend growth rate  $\lambda$

**Lucas's Model** We now describe a version of a celebrated asset pricing model of Robert E. Lucas, Jr. [Luc78]

We present only an outline, with full derivations left to a later lecture

In a nutshell, Lucas made two key assumptions to specialize our general asset pricing equation (2.167):

- He assumed that the stochastic discount factor took the form

$$m_{t+1} = \beta \frac{u'(c_{t+1})}{u'(c_t)}$$

where  $u$  is a concave utility function and  $c_t$  is time  $t$  consumption of a representative consumer

- He assumed that the consumption process  $\{c_t\}_{t=0}^{\infty}$  of the representative consumer is governed by a Markov process
- He assumed that the asset being priced is a claim on the aggregate consumption process, so that  $d_t = c_t$

It is common to assume that the utility function takes the following form

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \text{ with } \gamma > 0 \quad (2.172)$$

where  $u(c) = \ln c$  when  $\gamma = 1$

The **constant relative risk aversion** (CRRA) utility function (2.172) implies that

$$u'(c) = c^{-\gamma}$$

so that

$$m_{t+1} = \beta \left( \frac{c_{t+1}}{c_t} \right)^{-\gamma}$$

or  $m_{t+1} = \beta \lambda_{t+1}^{-\gamma}$  when

$$\lambda_{t+1} := \left( \frac{c_{t+1}}{c_t} \right) \quad (2.173)$$

Below, we'll often assume that the gross growth rate of consumption  $\lambda_{t+1}$  is governed by a Markov process

### Classes of Assets

For the remainder of this lecture we focus on computing asset prices when

- endowments follow a finite state Markov chain
- agents are risk averse, and prices obey (2.167)

As in [MP85], there is an endowment of a consumption good that follows

$$c_{t+1} = \lambda_{t+1} c_t \quad (2.174)$$

Here  $\lambda_t$  is governed by the  $n$  state Markov chain discussed *above*

A *Lucas tree* is a claim on this endowment stream

We'll price several distinct assets, including

- The Lucas tree itself
- A consol (a type of bond issued by the UK government in the 19th century)
- Finite and infinite horizon call options on a consol

**Pricing the Lucas tree** Using (2.167), the definition of  $u$  and (2.174) leads to

$$p_t = \mathbb{E}_t \left[ \beta \lambda_{t+1}^{-\gamma} (c_{t+1} + p_{t+1}) \right] \quad (2.175)$$

Drawing intuition from *our earlier discussion* on pricing with Markov growth, we guess a pricing function of the form  $p_t = v(\lambda_t)c_t$  where  $v$  is yet to be determined

If we substitute this guess into (2.175) and rearrange, we obtain

$$v(\lambda_t)c_t = \mathbb{E}_t \left[ \beta \lambda_{t+1}^{-\gamma} (c_{t+1} + c_{t+1}v(\lambda_{t+1})) \right]$$

Using (2.174) again and simplifying gives

$$v(\lambda_t) = \mathbb{E}_t \left[ \beta \lambda_{t+1}^{1-\gamma} (1 + v(\lambda_{t+1})) \right]$$

As before we let  $v(s_i) = v_i$ , so that  $v$  is modeled as an  $n \times 1$  vector, and

$$v_i = \beta \sum_{j=1}^n P_{ij} s_j^{1-\gamma} (1 + v_j) \quad (2.176)$$

Letting  $\tilde{P}_{ij} = P_{ij} s_j^{1-\gamma}$ , we can write (2.176) as  $v = \beta \tilde{P} \mathbf{1} + \beta \tilde{P} v$

Assuming again that the eigenvalues of  $\tilde{P}$  are strictly less than  $\beta^{-1}$  in modulus, we can solve this to yield

$$v = \beta(I - \beta \tilde{P})^{-1} \tilde{P} \mathbf{1} \quad (2.177)$$

With log preferences, we have  $\gamma = 1$  and hence  $s^{1-\gamma} = \mathbf{1}$

Recalling that  $P^i \mathbf{1} = \mathbf{1}$  for all  $i$  and applying *Neumann's geometric series lemma*, we are led to

$$v = \beta(I - \beta P)^{-1} \mathbf{1} = \beta \sum_{i=0}^{\infty} \beta^i P^i \mathbf{1} = \beta \frac{1}{1 - \beta} \mathbf{1}$$

Thus, with log preferences, the price-dividend ratio for a Lucas tree is constant

**A Risk-Free Consol** Consider the same pure exchange representative agent economy

A risk-free consol promises to pay a constant amount  $\zeta > 0$  each period

Recycling notation, let  $p_t$  now be the price of an ex-coupon claim to the consol

An ex-coupon claim to the consol entitles the owner at the end of period  $t$  to

- $\zeta$  in period  $t + 1$ , plus
- the right to sell the claim for  $p_{t+1}$  next period

The price satisfies

$$u'(c_t)p_t = \beta \mathbb{E}_t [u'(c_{t+1})(\zeta + p_{t+1})]$$

Substituting  $u'(c) = c^{-\gamma}$  into the above equation yields

$$c_t^{-\gamma} p_t = \beta \mathbb{E}_t [c_{t+1}^{-\gamma} (\zeta + p_{t+1})] = \beta c_t^{-\gamma} \mathbb{E}_t [\lambda_{t+1}^{-\gamma} (\zeta + p_{t+1})]$$

It follows that

$$p_t = \beta \mathbb{E}_t [\lambda_{t+1}^{-\gamma} (\zeta + p_{t+1})] \quad (2.178)$$

Now guess that the price takes the form

$$p_t = p(\lambda_t) = p_i \quad \text{when } \lambda_t = s_i$$

Then (2.178) becomes

$$p_i = \beta \sum_j P_{ij} s_j^{-\gamma} (\zeta + p_j)$$

which can be expressed as  $p = \beta \check{P} \zeta \mathbf{1} + \beta \check{P} p$ , or

$$p = \beta(I - \beta \check{P})^{-1} \check{P} \zeta \mathbf{1} \quad (2.179)$$

where  $\check{P}_{ij} = P_{ij} s_j^{-\gamma}$

**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

Thus, the stochastic discount factor continues to be  $m_{t+1} = \beta \frac{u'(c_{t+1})}{u'(c_t)}$  and the consumption growth rate continues to be  $\lambda_{t+1}$ , which is governed by a finite state Markov chain.

Let  $w(\lambda_t, p_S)$  be the value of the option when the time  $t$  growth state is known to be  $\lambda_t$  but *before* the owner has decided whether or not to exercise the option at time  $t$  (i.e., today)

Recalling that  $p(\lambda_t)$  is the value of the consol when the initial growth state is  $\lambda_t$ , the value of the option satisfies

$$w(\lambda_t, p_S) = \max \left\{ \beta \mathbb{E}_t \frac{u'(c_{t+1})}{u'(c_t)} w(\lambda_{t+1}, p_S), p(\lambda_t) - p_S \right\}$$

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

$$w(s_i, p_S) = \max \left\{ \beta \sum_{j=1}^n P_{ij} s_j^{-\gamma} w(s_j, p_S), p(s_i) - p_S \right\} \quad (2.180)$$

Letting  $\hat{P}_{ij} = P_{ij} s_j^{-\gamma}$  and  $w_i = w(s_i, p_S)$ , we can express (2.180) as the nonlinear vector equation

$$w = \max \{ \beta \hat{P}w, p - p_S \mathbf{1} \} \quad (2.181)$$

To solve (2.181), form the operator  $T$  mapping vector  $w$  into vector  $Tw$  via

$$Tw = \max \{ \beta \hat{P}w, p - p_S \mathbf{1} \}$$

Start at some initial  $w$  and iterate to convergence with  $T$

**Finite-horizon options** Finite horizon options obey functional equations closely related to (2.180)

A  $k$  period option expires after  $k$  periods

At time  $t$ , 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  $t, t+1, \dots, t+k-1$

The option expires at time  $t+k$

Thus, for  $k = 1, 2, \dots$ , let  $w(s_i, k)$  be the value of a  $k$ -period option

It obeys

$$w(s_i, k) = \max \left\{ \beta \sum_{j=1}^n P_{ij} s_j^{-\gamma} w(s_j, k-1), p(s_i) - p_S \right\}$$

where  $w(s_i, 0) = 0$  for all  $i$

We can express the preceding as the sequence of nonlinear vector equations

$$w_i^{(k)} = \max \left\{ \beta \sum_{j=1}^n \hat{P}_{ij} w_j^{(k-1)}, p_i - p_S \right\}, \quad k = 1, 2, \dots \quad \text{with } w^0 = 0$$

**Other Prices** Let's look at the pricing of several other assets

**The one-period risk-free interest rate** For this economy, the stochastic discount factor is

$$m_{t+1} = \beta \frac{c_{t+1}^{-\gamma}}{c_t^{-\gamma}} = \beta \lambda_{t+1}^{-\gamma}$$

It follows that the reciprocal  $R_t^{-1}$  of the gross risk-free interest rate  $R_t$  is

$$\mathbb{E}_t m_{t+1} = \beta \sum_{j=1}^n P_{ij} s_j^{-\gamma}$$

or

$$m_1 = \beta P s^{-\gamma}$$

where the  $i$ -th element of  $m_1$  is the reciprocal of the one-period gross risk-free interest rate when  $\lambda_t = s_i$

**$j$  period risk-free interest rates** 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 when  $\lambda_t = s_i$

Again, let  $\hat{P}_{ij} = P_{ij} s_j^{-\gamma}$

Then  $m_1 = \beta \hat{P}$ , and  $m_{j+1} = \hat{P} m_j$  for  $j \geq 1$

### Implementation

The class `AssetPrices` from the `QuantEcon.applications` package provides methods for computing some of the prices described above

We print the code here for convenience

```
"""
Filename: asset_pricing.py

Computes asset prices in a Lucas endowment economy when the endowment obeys
geometric growth driven by a finite state Markov chain. That is,

.. math::
    d_{\{t+1\}} = X_{\{t+1\}} d_t

where :math:`X_t` is a finite Markov chain with transition matrix P.

References
-----
    http://quant-econ.net/py/markov_asset.html

"""

import numpy as np
from numpy.linalg import solve
```

```

class AssetPriceModel:
    r"""
    A class that stores the primitives of the asset pricing model, plus a few
    useful matrices as attributes

    Parameters
    -----
    beta : scalar, float
        Discount factor
    mc : MarkovChain
        Contains the transition matrix and set of state values for the state
        proces
    gamma : scalar(float)
        Coefficient of risk aversion

    Attributes
    -----
    beta, mc, gamm : as above

    P_tilde : ndarray
        The matrix :math:`P(x, y) y^{(1 - \gamma)}`

    P_tilde : ndarray
        The matrix :math:`P(x, y) y^{(\gamma)}`

    """
    def __init__(self, beta, mc, gamma):
        self.beta, self.mc, self.gamma = beta, mc, gamma
        self.n = self.mc.P.shape[0]

    @property
    def P_tilde(self):
        P = self.mc.P
        y = self.mc.state_values
        return P * y**(1 - self.gamma) # using broadcasting

    @property
    def P_check(self):
        P = self.mc.P
        y = self.mc.state_values
        return P * y**(-self.gamma) # using broadcasting

    def tree_price(apm):
        """
        Computes the price-dividend ratio of the Lucas tree.

        Parameters
        -----
        apm: AssetPriceModel
            An instance of AssetPriceModel containing primitives

```

```

    Returns
    -----
    v : array_like(float)
        Lucas tree price-dividend ratio

    """
    # == Simplify names == #
    beta = apm.beta
    P_tilde = apm.P_tilde

    # == Compute v == #
    I = np.identity(apm.n)
    O = np.ones(apm.n)
    v = beta * solve(I - beta * P_tilde, P_tilde @ O)

    return v

def consol_price(apm, zeta):
    """
    Computes price of a consol bond with payoff zeta

    Parameters
    -----
    apm: AssetPriceModel
        An instance of AssetPriceModel containing primitives

    zeta : scalar(float)
        Coupon of the console

    Returns
    -----
    p_bar : array_like(float)
        Console bond prices

    """
    # == Simplify names == #
    beta = apm.beta

    # == Compute price == #
    P_check = apm.P_check
    I = np.identity(apm.n)
    O = np.ones(apm.n)
    p_bar = beta * solve(I - beta * P_check, P_check.dot(zeta * O))

    return p_bar

def call_option(apm, zeta, p_s, T=[], epsilon=1e-8):
    """
    Computes price of a call option on a consol bond, both finite
    and infinite horizon

```

```

Parameters
-----
apm: AssetPriceModel
    An instance of AssetPriceModel containing primitives

zeta : scalar(float)
    Coupon of the console

p_s : scalar(float)
    Strike price

T : iterable(integers)
    Length of option in the finite horizon case

epsilon : scalar(float), optional(default=1e-8)
    Tolerance for infinite horizon problem

Returns
-----
w_bar : array_like(float)
    Infinite horizon call option prices

w_bars : dict
    A dictionary of key-value pairs {t: vec}, where t is one of
    the dates in the list T and vec is the option prices at that
    date

"""
# == Simplify names, initialize variables ==
beta = apm.beta
P_check = apm.P_check

# == Compute consol price ==
v_bar = consol_price(apm, zeta)

# == Compute option price ==
w_bar = np.zeros(apm.n)
error = epsilon + 1
t = 0
w_bars = {}
while error > epsilon:
    if t in T:
        w_bars[t] = w_bar

    # == Maximize across columns ==
    to_stack = (beta*P_check.dot(w_bar), v_bar-p_s)
    w_bar_new = npamax(np.vstack(to_stack), axis=0)

    # == Find maximal difference of each component ==
    error = npamax(np.abs(w_bar-w_bar_new))

    # == Update ==
    w_bar = w_bar_new

```

```
t += 1

return w_bar, w_bars
```

## Exercises

**Exercise 1** Compute the price of the Lucas tree in an economy with 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([1.05, 1.025, 1.0, 0.975, 0.95]) # state values
gamma = 2.0
beta = 0.94
zeta = 1.0
```

Using the same set of primitives, compute the price of the risk-free console when  $\zeta = 1$

Do the same for the call option on the console when  $p_S = 150.0$

Compute the value of the option at dates  $T = [10, 20, 30]$

## Solutions

[Solution notebook](#)

# A Harrison-Kreps (1978) Model of Asset Prices

## Contents

- *A Harrison-Kreps (1978) 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

All code you see below can be obtained from the `harrison_kreps` directory of the `QuantEcon.applications` repository

**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:

```
In [1]: import numpy as np
In [2]: import quantecon as qe
In [3]: qa = np.array([[1/2, 1/2], [2/3, 1/3]])
In [4]: qb = np.array([[2/3, 1/3], [1/4, 3/4]])
In [5]: mcA = qe.MarkovChain(qa)
In [6]: mcB = qe.MarkovChain(qb)
In [7]: mcA.stationary_distributions
Out[7]: array([[ 0.57142857,  0.42857143]])
In [8]: mcB.stationary_distributions
Out[8]: array([[ 0.42857143,  0.57142857]])
```

The stationary distribution of  $P_a$  is approximately  $\pi_B = [.57 \quad .43]$

The stationary distribution of  $P_b$  is approximately  $\pi_B = [.43 \quad .57]$

**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 <sup>1</sup>
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

---

<sup>1</sup> 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.

- 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  $\pi_A = [.57 \quad .43]$  and  $\pi_B = [.43 \quad .57]$  tell us that a type  $B$  person is more optimistic about the dividend process in the long run than is a type A person

**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  $P_a, P_b, \beta$

Here

- $p_a$  is the equilibrium price function under homogeneous beliefs  $P_a$
- $p_b$  is the equilibrium price function under homogeneous beliefs  $P_b$

- $\bar{p}_a$  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:

$$p_h(s) = \beta (P_h(s, 1)p_i(1) + P_h(s, 2)(1 + p_h(2))), \quad s = 1, 2$$

These equations imply that the equilibrium price vector is

$$\begin{bmatrix} p_h(1) \\ p_h(2) \end{bmatrix} = \beta[I - \beta P_h]^{-1} P_h \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad (2.182)$$

The first two rows of the table report  $p_a(s)$  and  $p_b(s)$

Here's a function that can be used to compute these values

```
"""
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
```

**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:

$$\bar{p}(s) = \beta \max \{ P_a(s, 1)\bar{p}(1) + P_a(s, 2)(1 + \bar{p}(2)), P_b(s, 1)\bar{p}(1) + P_b(s, 2)(1 + \bar{p}(2)) \} \quad (2.183)$$

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 (2.183) is a functional equation that, like a Bellman equation, can be solved by

- starting with a guess for the price vector  $\bar{p}$  and
- iterating to convergence on the operator that maps a guess  $\bar{p}^j$  into an updated guess

$\bar{p}^{j+1}$  defined by the right side of (2.183), namely

$$\bar{p}^{j+1}(s) = \beta \max \left\{ P_a(s, 1)\bar{p}^j(1) + P_a(s, 2)(1 + \bar{p}^j(2)), P_b(s, 1)\bar{p}^j(1) + P_b(s, 2)(1 + \bar{p}^j(2)) \right\} \quad (2.184)$$

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 converge
    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 succeed 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
```

**Insufficient Funds** Outcomes differ when the more optimistic type of investor has insufficient wealth — or insufficient ability to borrow enough — to hold the entire stock of the asset

In this case, the asset price must adjust to attract pessimistic investors

Instead of equation (2.183), the equilibrium price satisfies

$$\check{p}(s) = \beta \min \{ P_a(s, 1)\check{p}(1) + P_a(s, 2)(1 + \check{p}(2)), P_b(s, 1)\check{p}(1) + P_b(s, 2)(1 + \check{p}(2)) \} \quad (2.185)$$

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 (2.185), optimistic investors think that the asset is underpriced

If they could, optimistic investors would willingly borrow at the one-period gross interest rate  $\beta^{-1}$  to purchase more of the asset

Implicit constraints on leverage prohibit them from doing so

When optimistic investors price the asset as in equation (2.183), 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 succeed in converging, break out of for loop
if np.max(np.sqrt((p_new - p_old)**2)) < 1e-12:
    break

return p_new

```

**Further Interpretation** [Sch14] interprets the Harrison-Kreps model as a model of a bubble — a situation in which an asset price exceeds what every investor thinks is merited by the 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 Permanent Income Model

### Contents

- *The Permanent Income Model*
  - *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

## 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

$$\mathbb{E}_t[X_{t+1}] = X_t, \quad t = 0, 1, 2, \dots$$

Here  $\mathbb{E}_t := \mathbb{E}[\cdot | \mathcal{F}_t]$  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  $\mathcal{F}_t = \{X_t, X_{t-1}, \dots, X_0\}$

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

$$\mathbb{E}_0 \left[ \sum_{t=0}^{\infty} \beta^t u(c_t) \right] \quad (2.186)$$

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

$$b_{t+1} = (1 + r)(c_t + b_t - y_t) \quad t \geq 0 \quad (2.187)$$

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  $t$
- $b_0$  is a given initial condition

**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

$$x_{t+1} = Ax_t + Cw_{t+1} \quad (2.188)$$

$$y_t = Ux_t \quad (2.189)$$

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  $x_t$ .

The restriction on  $\rho(A)$  prevents income from growing so fast that some discounted geometric sums of some infinite sequences below become infinite

We also impose the *no Ponzi scheme* condition

$$\mathbb{E}_0 \left[ \sum_{t=0}^{\infty} \beta^t b_t^2 \right] < \infty \quad (2.190)$$

This condition rules out an always-borrow scheme that would allow the household to enjoy unbounded or bliss consumption forever

Regarding preferences, we assume the quadratic utility function

$$u(c_t) = -(c_t - \bar{c})^2$$

where  $\bar{c}$  is a bliss level of consumption

(Along with this quadratic utility specification, we allow consumption to be negative)

**First Order Conditions** First-order conditions for maximizing (2.186) subject to (2.187) are

$$\mathbb{E}_t[u'(c_{t+1})] = u'(c_t), \quad t = 0, 1, \dots \quad (2.191)$$

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, (2.191) has the striking implication that consumption follows a martingale:

$$\mathbb{E}_t[c_{t+1}] = c_t \quad (2.192)$$

(In fact quadratic preferences are *necessary* for this conclusion <sup>1</sup>)

One way to interpret (2.192) is that consumption will only change when "new information" about permanent income is revealed

These ideas will be clarified below

**The Optimal Decision Rule** The *state* vector confronting the household at  $t$  is  $[b_t \ x_t]$

Here

- $x_t$  is an *exogenous* component, unaffected by household behavior
- $b_t$  is an *endogenous* component (since it depends on the decision rule)

Note that  $x_t$  contains all variables useful for forecasting the household's future endowment

Now let's deduce the optimal decision rule <sup>2</sup>

---

**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

---

<sup>1</sup> A linear marginal utility is essential for deriving (2.192) from (2.191). 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 (2.191). 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 (2.191) 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  $+\infty$ .

<sup>2</sup> An optimal decision rule is a map from current state into current actions—in this case, consumption

We want to solve the system of difference equations formed by (2.187) and (2.192) subject to the boundary condition (2.190)

To accomplish this, observe first that (2.190) implies  $\lim_{t \rightarrow \infty} \beta^t b_{t+1} = 0$

Using this restriction on the debt path and solving (2.187) forward yields

$$b_t = \sum_{j=0}^{\infty} \beta^j (y_{t+j} - c_{t+j}) \quad (2.193)$$

Take conditional expectations on both sides of (2.193) and use the *law of iterated expectations* to deduce

$$b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - \frac{c_t}{1-\beta} \quad (2.194)$$

Expressed in terms of  $c_t$  we get

$$c_t = (1-\beta) \left[ \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - b_t \right] \quad (2.195)$$

If we define the *net rate of interest*  $r$  by  $\beta = \frac{1}{1+r}$ , we can also express this equation as

$$c_t = \frac{r}{1+r} \left[ \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - b_t \right]$$

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

**A State-Space Representation** The preceding results provide a decision rule and hence the dynamics of both state and control variables

First note that equation (2.195) represents  $c_t$  as a function of the state  $[b_t \ x_t]$  confronting the household

If the last statement isn't clear, recall that  $\mathbb{E}_t[y_{t+j}]$  can be expressed as a function of  $x_t$ , since the latter contains all information useful for forecasting the household's endowment process

In fact, from *this discussion* we see that

$$\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] = \mathbb{E}_t \left[ \sum_{j=0}^{\infty} \beta^j y_{t+j} \right] = U(I - \beta A)^{-1} x_t$$

Using this expression, we can obtain a linear state-space system governing consumption, debt and income:

$$x_{t+1} = Ax_t + Cw_{t+1} \quad (2.196)$$

$$b_{t+1} = b_t + U[(I - \beta A)^{-1}(A - I)]x_t \quad (2.197)$$

$$y_t = Ux_t \quad (2.198)$$

$$c_t = (1 - \beta)[U(I - \beta A)^{-1}x_t - b_t] \quad (2.199)$$

Define

$$z_t = \begin{bmatrix} x_t \\ b_t \end{bmatrix}, \quad \tilde{A} = \begin{bmatrix} A & 0 \\ U(I - \beta A)^{-1}(A - I) & 1 \end{bmatrix}, \quad \tilde{C} = \begin{bmatrix} C \\ 0 \end{bmatrix}$$

and

$$\tilde{U} = \begin{bmatrix} U & 0 \\ (1 - \beta)U(I - \beta A)^{-1} & -(1 - \beta) \end{bmatrix}, \quad \tilde{y}_t = \begin{bmatrix} y_t \\ b_t \end{bmatrix}$$

Then we can express equation (2.196) as

$$z_{t+1} = \tilde{A}z_t + \tilde{C}w_{t+1} \quad (2.200)$$

$$\tilde{y}_t = \tilde{U}z_t \quad (2.201)$$

We can use the following formulas from [state-space representation](#) to compute population mean  $\mu_t = \mathbb{E} z_t$  and covariance  $\Sigma_t := \mathbb{E} [(z_t - \mu_t)(z_t - \mu_t)']$

$$\mu_{t+1} = \tilde{A}\mu_t \quad \text{with } \mu_0 \text{ given} \quad (2.202)$$

$$\Sigma_{t+1} = \tilde{A}\Sigma_t\tilde{A}' + \tilde{C}\tilde{C}' \quad \text{with } \Sigma_0 \text{ given} \quad (2.203)$$

We can then compute the mean and covariance of  $\tilde{y}_t$  from

$$\mu_{y,t} = \tilde{U}\mu_t\Sigma_{y,t} = \tilde{U}\Sigma_t\tilde{U}' \quad (2.204)$$

**A Simple Example with iid Income** To gain some preliminary intuition on the implications of (2.196), 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

$$x_t = \begin{bmatrix} x_t^1 \\ 1 \end{bmatrix}, \quad A = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \quad U = [1 \quad \mu], \quad C = \begin{bmatrix} \sigma \\ 0 \end{bmatrix}$$

Finally, let  $b_0 = x_0^1 = 0$

Under these assumptions we have  $y_t = \mu + \sigma w_t \sim N(\mu, \sigma^2)$

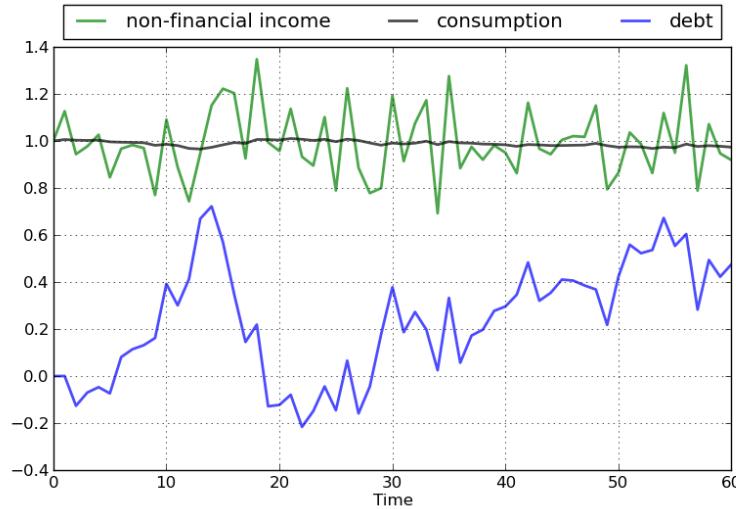
Further, if you work through the state space representation, you will see that

$$\begin{aligned} b_t &= -\sigma \sum_{j=1}^{t-1} w_j \\ c_t &= \mu + (1 - \beta)\sigma \sum_{j=1}^t w_j \end{aligned}$$

Thus income is iid and debt and consumption are both Gaussian random walks

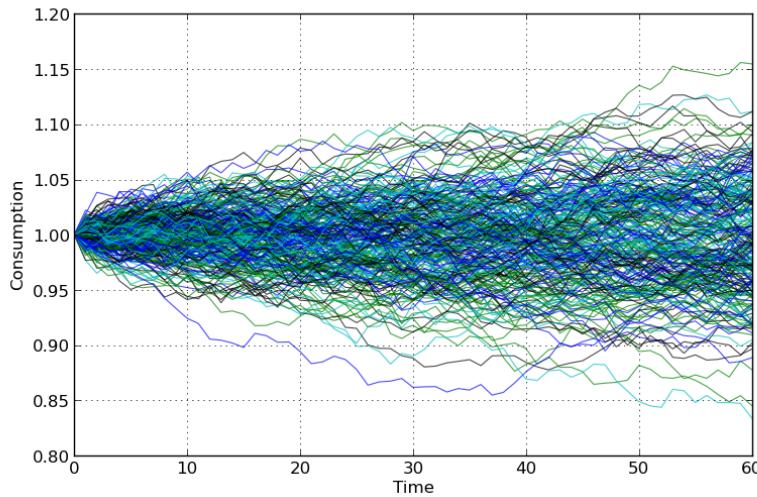
Defining assets as  $-b_t$ , we see that assets are just the cumulative sum of unanticipated income prior to the present date

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 (2.195) forward one period and eliminate  $b_{t+1}$  by using (2.187) to obtain

$$c_{t+1} = (1 - \beta) \sum_{j=0}^{\infty} \beta^j \mathbb{E}_{t+1}[y_{t+j+1}] - (1 - \beta) [\beta^{-1}(c_t + b_t - y_t)]$$

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

$$c_{t+1} - c_t = (1 - \beta) \sum_{j=0}^{\infty} \beta^j \{ \mathbb{E}_{t+1}[y_{t+j+1}] - \mathbb{E}_t[y_{t+j+1}] \} \quad (2.205)$$

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 (2.205) and (2.194), which is repeated here:

$$b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - \frac{1}{1 - \beta} c_t \quad (2.206)$$

Equation (2.206) 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

$$\begin{aligned} \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} &= U(I - \beta A)^{-1} x_t \\ \mathbb{E}_{t+1} \sum_{j=0}^{\infty} \beta^j y_{t+j+1} &= U(I - \beta A)^{-1} x_{t+1} \\ \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j+1} &= U(I - \beta A)^{-1} A x_t \end{aligned}$$

Using these formulas together with (2.188) and substituting into (2.205) and (2.206) gives the following representation for the consumer's optimum decision rule:

$$c_{t+1} = c_t + (1 - \beta) U(I - \beta A)^{-1} C w_{t+1} \quad (2.207)$$

$$b_t = U(I - \beta A)^{-1} x_t - \frac{1}{1 - \beta} c_t \quad (2.208)$$

$$y_t = U x_t \quad (2.209)$$

$$x_{t+1} = A x_t + C w_{t+1} \quad (2.210)$$

Representation (2.207) makes clear that

- The state can be taken as  $(c_t, x_t)$ 
  - The endogenous part is  $c_t$  and the exogenous part is  $x_t$
  - 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 (2.192)

**Cointegration** Representation (2.207) 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  $x_t$  is asymptotically stationary<sup>4</sup>

Despite this, both  $c_t$  and  $b_t$  will be non-stationary because they have unit roots (see (2.196) 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 (2.207) we have

$$(1 - \beta)b_t + c_t = (1 - \beta)U(I - \beta A)^{-1}x_t \quad (2.211)$$

Hence the linear combination  $(1 - \beta)b_t + c_t$  is asymptotically stationary

Accordingly, Granger and Engle would call  $[(1 - \beta) \ 1]$  a *cointegrating vector* for the state

When applied to the nonstationary vector process  $[b_t \ c_t]',$  it yields a process that is asymptotically stationary

Equation (2.211) can be arranged to take the form

$$(1 - \beta)b_t + c_t = (1 - \beta)\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j}, \quad (2.212)$$

Equation (2.212) asserts that the *cointegrating residual* on the left side equals the conditional expectation of the geometric sum of future incomes on the right<sup>6</sup>

**Cross-Sectional Implications** Consider again (2.207), this time in light of our discussion of distribution dynamics in the [lecture on linear systems](#)

The dynamics of  $c_t$  are given by

$$c_{t+1} = c_t + (1 - \beta)U(I - \beta A)^{-1}Cw_{t+1} \quad (2.213)$$

or

$$c_t = c_0 + \sum_{j=1}^t \hat{w}_j \quad \text{for } \hat{w}_{t+1} := (1 - \beta)U(I - \beta A)^{-1}Cw_{t+1}$$

<sup>4</sup> This would be the case if, for example, the *spectral radius* of  $A$  is strictly less than one

<sup>6</sup> See Campbell and Shiller (1988) and Lettau and Ludvigson (2001, 2004) for interesting applications of related ideas.

The unit root affecting  $c_t$  causes the time  $t$  variance of  $c_t$  to grow linearly with  $t$

In particular, since  $\{\hat{w}_t\}$  is iid, we have

$$\text{Var}[c_t] = \text{Var}[c_0] + t \hat{\sigma}^2 \quad (2.214)$$

when

$$\hat{\sigma}^2 := (1 - \beta)^2 U(I - \beta A)^{-1} C C' (I - \beta A')^{-1} U'$$

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 (2.214) 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 (2.188) has the moving average representation

$$y_{t+1} = d(L)w_{t+1} \quad (2.215)$$

where

- $d(L) = \sum_{j=0}^{\infty} d_j L^j$  for some sequence  $d_j$ , where  $L$  is the lag operator<sup>3</sup>
- at time  $t$ , the household has an information set<sup>5</sup>  $w^t = [w_t, w_{t-1}, \dots]$

Notice that

$$y_{t+j} - \mathbb{E}_t[y_{t+j}] = d_0 w_{t+j} + d_1 w_{t+j-1} + \dots + d_{j-1} w_{t+1}$$

It follows that

$$\mathbb{E}_{t+1}[y_{t+j}] - \mathbb{E}_t[y_{t+j}] = d_{j-1} w_{t+1} \quad (2.216)$$

Using (2.216) in (2.205) gives

$$c_{t+1} - c_t = (1 - \beta)d(\beta)w_{t+1} \quad (2.217)$$

The object  $d(\beta)$  is the *present value of the moving average coefficients* in the representation for the endowment process  $y_t$

---

<sup>3</sup> Representation (2.188) implies that  $d(L) = U(I - \beta A)^{-1}C$ .

<sup>5</sup> 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.

## Two Classic Examples

We illustrate some of the preceding ideas with the following two examples

In both examples, the endowment follows the process  $y_t = x_{1t} + x_{2t}$  where

$$\begin{bmatrix} x_{1t+1} \\ x_{2t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1t} \\ x_{2t} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} w_{1t+1} \\ w_{2t+1} \end{bmatrix}$$

Here

- $w_{t+1}$  is an iid  $2 \times 1$  process distributed as  $N(0, I)$
- $x_{1t}$  is a permanent component of  $y_t$
- $x_{2t}$  is a purely transitory component

**Example 1** Assume as before that the consumer observes the state  $x_t$  at time  $t$

In view of (2.207) we have

$$c_{t+1} - c_t = \sigma_1 w_{1t+1} + (1 - \beta) \sigma_2 w_{2t+1} \quad (2.218)$$

Formula (2.218) shows how an increment  $\sigma_1 w_{1t+1}$  to the permanent component of income  $x_{1t+1}$  leads to

- a permanent one-for-one increase in consumption and
- no increase in savings  $-b_{t+1}$

But the purely transitory component of income  $\sigma_2 w_{2t+1}$  leads to a permanent increment in consumption by a fraction  $1 - \beta$  of transitory income

The remaining fraction  $\beta$  is saved, leading to a permanent increment in  $-b_{t+1}$

Application of the formula for debt in (2.196) to this example shows that

$$b_{t+1} - b_t = -x_{2t} = -\sigma_2 w_{2t} \quad (2.219)$$

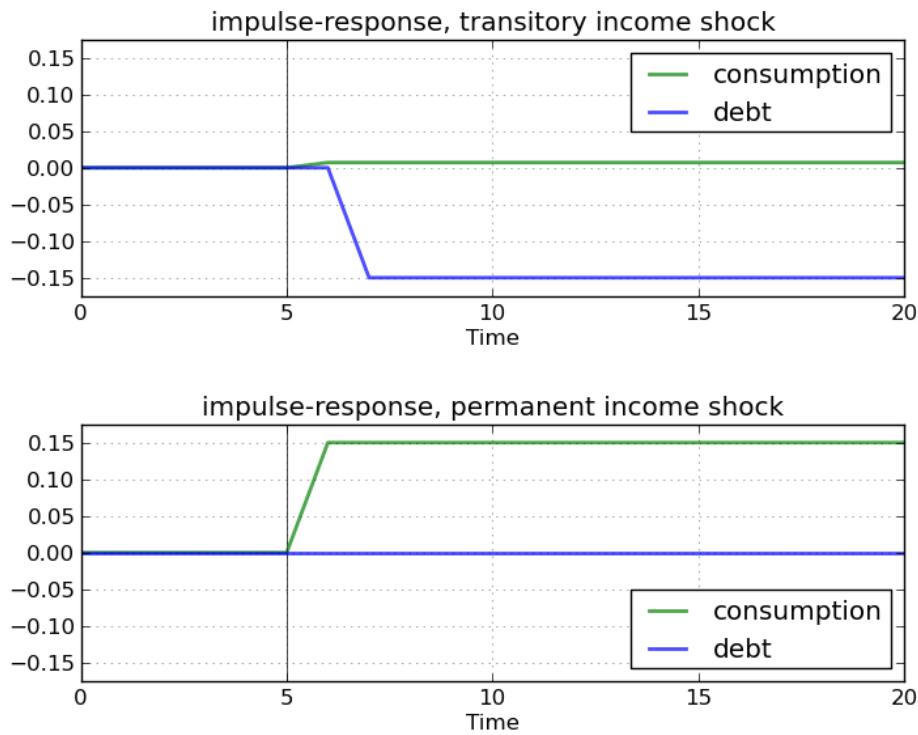
This confirms that none of  $\sigma_1 w_{1t}$  is saved, while all of  $\sigma_2 w_{2t}$  is saved

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_income/perm_inc_ir.py` from the [applications repository](#), 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_xlim(-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  $x_t$

Under this assumption, it is appropriate to use an *innovation representation* to form  $A, C, U$  in (2.207)

The discussion in sections 2.9.1 and 2.11.3 of [LS12] shows that the pertinent state space representation for  $y_t$  is

$$\begin{bmatrix} y_{t+1} \\ a_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & -(1-K) \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ a_t \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} a_{t+1}$$

$$y_t = [1 \ 0] \begin{bmatrix} y_t \\ a_t \end{bmatrix}$$

where

- $K :=$  the stationary Kalman gain
- $a_t := y_t - E[y_t | y_{t-1}, \dots, y_0]$

In the same discussion in [LS12] it is shown that  $K \in [0, 1]$  and that  $K$  increases as  $\sigma_1/\sigma_2$  does

In other words, as the ratio of the standard deviation of the permanent shock to that of the transitory shock increases

Applying formulas (2.207) implies

$$c_{t+1} - c_t = [1 - \beta(1 - K)]a_{t+1} \quad (2.220)$$

where the endowment process can now be represented in terms of the univariate innovation to  $y_t$  as

$$y_{t+1} - y_t = a_{t+1} - (1 - K)a_t \quad (2.221)$$

Equation (2.221) 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 (2.221), the first difference of income is a first-order moving average

Equation (2.220) asserts that the first difference of consumption is iid

Application of formula to this example shows that

$$b_{t+1} - b_t = (K - 1)a_t \quad (2.222)$$

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 (2.191) come from?

Here we'll give a proof for the two period case, which is representative of the general argument

The finite horizon equivalent of the no-Ponzi condition is that the agent cannot end her life in debt, so  $b_2 = 0$

From the budget constraint (2.187) 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 (2.191) in the two period case

The proof for the general case is not dissimilar

---

CHAPTER  
THREE

---

## ADVANCED APPLICATIONS

This advanced section of the course contains more complex applications, and can be read selectively, according to your interests

### 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,

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

Equivalently,

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

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

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  $p: S \times S \rightarrow \mathbb{R}$  with the property that

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

(Integrals are over the whole space unless otherwise specified)

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

$$p_w(x, y) := \frac{1}{\sqrt{2\pi}} \exp\left\{-\frac{(y-x)^2}{2}\right\} \quad (3.1)$$

What kind of model does  $p_w$  represent?

The answer is, the (normally distributed) random walk

$$X_{t+1} = X_t + \xi_{t+1} \quad \text{where} \quad \{\xi_t\} \stackrel{\text{IID}}{\sim} N(0, 1) \quad (3.2)$$

To see this, let's find the stochastic kernel  $p$  corresponding to (3.2)

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

Letting  $X_t = x$  in (3.2) and considering the distribution of  $X_{t+1}$ , we see that  $p(x, \cdot) = N(x, 1)$

In other words,  $p$  is exactly  $p_w$ , as defined in (3.1)

**Connection to Stochastic Difference Equations** In the previous section, we made the connection between stochastic difference equation (3.2) and stochastic kernel (3.1)

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

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

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

$$X_{t+1} = \mu(X_t) + \sigma(X_t) \xi_{t+1} \quad (3.3)$$

Here we assume that

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

**Example 1:** The random walk (3.2) is a special case of (3.3), with  $\mu(x) = x$  and  $\sigma(x) = 1$

**Example 2:** Consider the ARCH model

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

Alternatively, we can write the model as

$$X_{t+1} = \alpha X_t + (\beta + \gamma X_t^2)^{1/2} \xi_{t+1} \quad (3.4)$$

This is a special case of (3.3) with  $\mu(x) = \alpha x$  and  $\sigma(x) = (\beta + \gamma x^2)^{1/2}$  **Example 3:** With stochastic production and a constant savings rate, the one-sector neoclassical growth model leads to a law of motion for capital per worker such as

$$k_{t+1} = s A_{t+1} f(k_t) + (1 - \delta) k_t \quad (3.5)$$

Here

- $s$  is the rate of savings
- $A_{t+1}$  is a production shock
  - The  $t + 1$  subscript indicates that  $A_{t+1}$  is not visible at time  $t$
- $\delta$  is a depreciation rate
- $f: \mathbb{R}_+ \rightarrow \mathbb{R}_+$  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 (3.5) is a special case of (3.3) with  $\mu(x) = (1 - \delta)x$  and  $\sigma(x) = sf(x)$

Now let's obtain the stochastic kernel corresponding to the generic model (3.3)

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

$$f_V(v) = \frac{1}{b} f_U\left(\frac{v-a}{b}\right) \quad (3.6)$$

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

Taking (3.6) as given for the moment, we can obtain the stochastic kernel  $p$  for (3.3) by recalling that  $p(x, \cdot)$  is the conditional density of  $X_{t+1}$  given  $X_t = x$

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

Hence, by (3.6),

$$p(x, y) = \frac{1}{\sigma(x)} \phi\left(\frac{y - \mu(x)}{\sigma(x)}\right) \quad (3.7)$$

For example, the growth model in (3.5) has stochastic kernel

$$p(x, y) = \frac{1}{sf(x)} \phi \left( \frac{y - (1 - \delta)x}{sf(x)} \right) \quad (3.8)$$

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

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

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

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

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

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

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

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

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

$$\psi_{t+1}(y) = \int p(x, y) \psi_t(x) dx, \quad \forall y \in S \quad (3.9)$$

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

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

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

$$(\psi P)(y) = \int p(x, y) \psi(x) dx \quad (3.10)$$

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

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

With this notation, we can write (3.9) more succinctly as  $\psi_{t+1}(y) = (\psi_t P)(y)$  for all  $y$ , or, dropping the  $y$  and letting “=” indicate equality of functions,

$$\psi_{t+1} = \psi_t P \quad (3.11)$$

Equation (3.11) tells us that if we specify a distribution for  $\psi_0$ , then the entire sequence of future distributions can be obtained by iterating with  $P$

It's interesting to note that (3.11) is a deterministic difference equation

Thus, by converting a stochastic difference equation such as (3.3) into a stochastic kernel  $p$  and hence an operator  $P$ , we convert a stochastic difference equation into a deterministic one (albeit in a much higher dimensional space)

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

**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 (3.10) and (3.11) using numerical integration

However, to produce  $\psi P$  from  $\psi$  via (3.10), you would need to integrate at every  $y$ , and there is a continuum of such  $y$

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

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

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

$$k_{t+1} = sA_{t+1}f(k_t) + (1 - \delta)k_t \quad (3.12)$$

Our aim is to compute the sequence  $\{\psi_t\}$  associated with this model and fixed initial condition  $\psi_0$

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

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

1. draw  $k_0$  from the specified initial condition  $\psi_0$
2. draw the shocks  $A_1, \dots, A_t$  from their specified density  $\phi$
3. compute  $k_t$  iteratively via (3.12)

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

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

A naive approach would be to use a histogram, or perhaps a [smoothed histogram](#) using SciPy's `gaussian_kde` function

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

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

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

$$\psi_t^n(y) = \frac{1}{n} \sum_{i=1}^n p(k_{t-1}^i, y) \quad (3.13)$$

where  $p$  is the growth model stochastic kernel in (3.8)

What is the justification for this slightly surprising estimator?

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

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

with probability one as  $n \rightarrow \infty$

Here the first equality is by the definition of  $\psi_{t-1}$ , and the second is by (3.9)

We have just shown that our estimator  $\psi_t^n(y)$  in (3.13) converges almost surely to  $\psi_t(y)$ , which is just what we want to compute

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

**Implementation** A class called LAE for estimating densities by this technique can be found in QuantEcon

We repeat it here for convenience

```
"""
Filename: lae.py

Authors: Thomas J. Sargent, John Stachurski,

Computes a sequence of marginal densities for a continuous state space
Markov chain :math:`X_t` where the transition probabilities can be represented
as densities. The estimate of the marginal density of :math:`X_t` is

.. math::

    \frac{1}{n} \sum_{i=0}^n p(X_{t-1}^i, y)

This is a density in y.

References
-----
http://quant-econ.net/py/stationary_densities.html

"""

from textwrap import dedent
import numpy as np
```

```

class LAE(object):
    """
    An instance is a representation of a look ahead estimator associated
    with a given stochastic kernel  $p$  and a vector of observations  $X$ .

    Parameters
    -----
    p : function
        The stochastic kernel. A function  $p(x, y)$  that is vectorized in
        both  $x$  and  $y$ 
    X : array_like(float)
        A vector containing observations

    Attributes
    -----
    p, X : see Parameters

    Examples
    -----
    >>> psi = LAE(p, X)
    >>> y = np.linspace(0, 1, 100)
    >>> psi(y) # Evaluate look ahead estimate at grid of points y

    """
    def __init__(self, p, X):
        X = X.flatten() # So we know what we're dealing with
        n = len(X)
        self.p, self.X = p, X.reshape((n, 1))

    def __repr__(self):
        return self.__str__()

    def __str__(self):
        m = """\
Look ahead estimator
- number of observations : {n}
"""
        return dedent(m.format(n=self.X.size))

    def __call__(self, y):
        """
        A vectorized function that returns the value of the look ahead
        estimate at the values in the array y.

        Parameters
        -----
        y : array_like(float)
            A vector of points at which we wish to evaluate the look-
            ahead estimator

        Returns
        """

```

```

psi_vals : array_like(float)
    The values of the density estimate at the points in y

"""
k = len(y)
v = self.p(self.X, y.reshape((1, k)))
psi_vals = np.mean(v, axis=0)      # Take mean along each row

return psi_vals.flatten()

```

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 (3.13) using

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

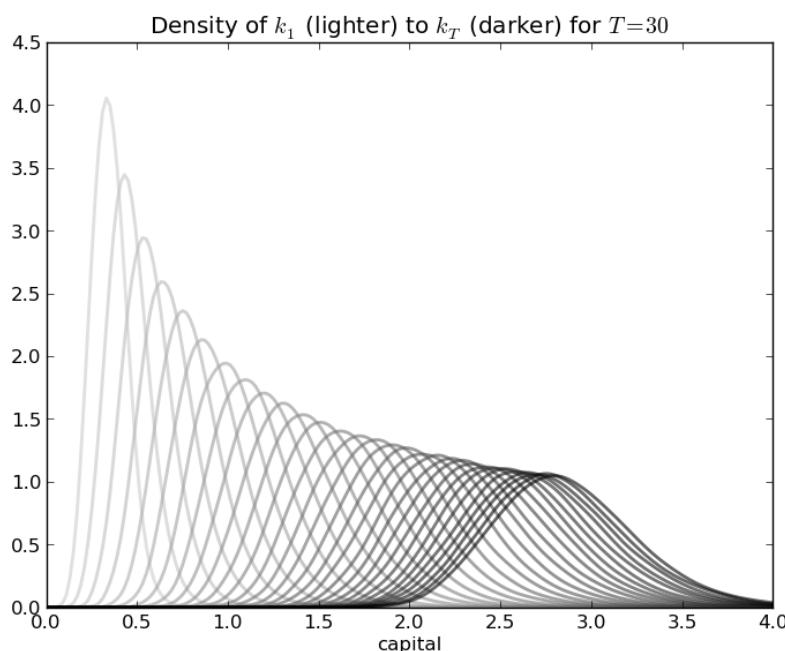
The function is vectorized, in the sense that if  $\psi$  is such an instance and  $y$  is an array, then the call  $\psi(y)$  acts elementwise

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

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

**Example** An example of usage for the stochastic growth model *described above* can be found in `stationary_densities/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

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

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

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

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

Hence it cannot be represented as a density

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

To this end, set

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

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

The distribution dynamics in (3.9) are then replaced by

$$F_{t+1}(y) = \int G(x, y) F_t(dx) \tag{3.14}$$

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

The intuition behind (3.14) is essentially the same as for (3.9)

**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 (3.10), a density  $\psi^*$  on  $S$  is called *stationary* for  $P$  if it is a fixed point of the operator  $P$

In other words,

$$\psi^*(y) = \int p(x, y)\psi^*(x) dx, \quad \forall y \in S \quad (3.15)$$

As with the finite case, if  $\psi^*$  is stationary for  $P$ , and the distribution of  $X_0$  is  $\psi^*$ , then, in view of (3.11),  $X_t$  will have this same distribution for all  $t$

Hence  $\psi^*$  is the stochastic equivalent of a steady state

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

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

Even existence can fail very easily

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

However, there are well-known conditions under which a stationary density  $\psi^*$  exists

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

$$\forall \psi \in \mathcal{D}, \quad \psi P^t \rightarrow \psi^* \quad \text{as} \quad t \rightarrow \infty \quad (3.16)$$

This combination of existence, uniqueness and global convergence in the sense of (3.16) is often referred to as *global stability*

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

$$\frac{1}{n} \sum_{t=1}^n h(X_t) \rightarrow \int h(x) \psi^*(x) dx \quad \text{as } n \rightarrow \infty \quad (3.17)$$

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

Note that the convergence in (3.17) does not depend on the distribution (or value) of  $X_0$

This is actually very important for simulation — it means we can learn about  $\psi^*$  (i.e., approximate the right hand side of (3.17) via the left hand side) without requiring any special knowledge about what to do with  $X_0$

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

In essence, it must be the case that

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

For one such set of conditions see theorem 8.2.14 of [EDTC](#)

In addition

- [\[SLP89\]](#) contains a classic (but slightly outdated) treatment of these topics
- From the mathematical literature, [\[LM94\]](#) and [\[MT09\]](#) give outstanding in depth treatments
- Section 8.1.2 of [EDTC](#) provides detailed intuition, and section 8.3 gives additional references
- [EDTC](#), section 11.3.4 provides a specific treatment for the growth model we considered in this lecture

**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 (3.16) — by simulating the path of densities from various initial conditions

Here is such a figure

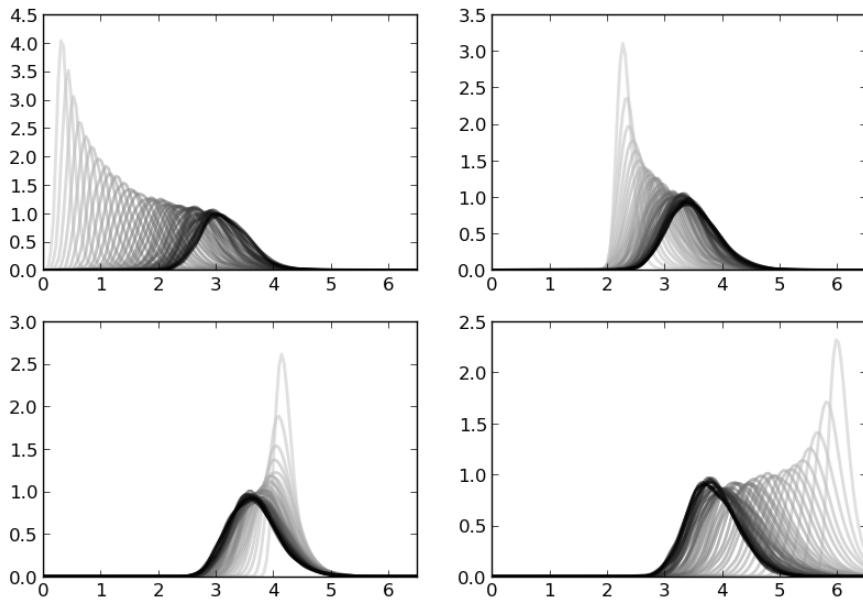
All sequences are converging towards the same limit, regardless of their initial condition

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

**Computing Stationary Densities** In the preceding figure, each sequence of densities is converging towards the unique stationary density  $\psi^*$

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

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



Let's say that we have a model of the form (3.3) that is stable and ergodic

Let  $p$  be the corresponding stochastic kernel, as given in (3.7)

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

$$\psi_n^*(y) = \frac{1}{n} \sum_{t=1}^n p(X_t, y) \quad (3.18)$$

This is essentially the same as the look ahead estimator (3.13), except that now the observations we generate are a single time series, rather than a cross section

The justification for (3.18) is that, with probability one as  $n \rightarrow \infty$ ,

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

where the convergence is by (3.17) and the equality on the right is by (3.15)

The right hand side is exactly what we want to compute

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

The first exercise helps illustrate this point

### Exercises

**Exercise 1** Consider the simple threshold autoregressive model

$$X_{t+1} = \theta |X_t| + (1 - \theta^2)^{1/2} \xi_{t+1} \quad \text{where } \{\xi_t\} \stackrel{\text{IID}}{\sim} N(0, 1) \quad (3.19)$$

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

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

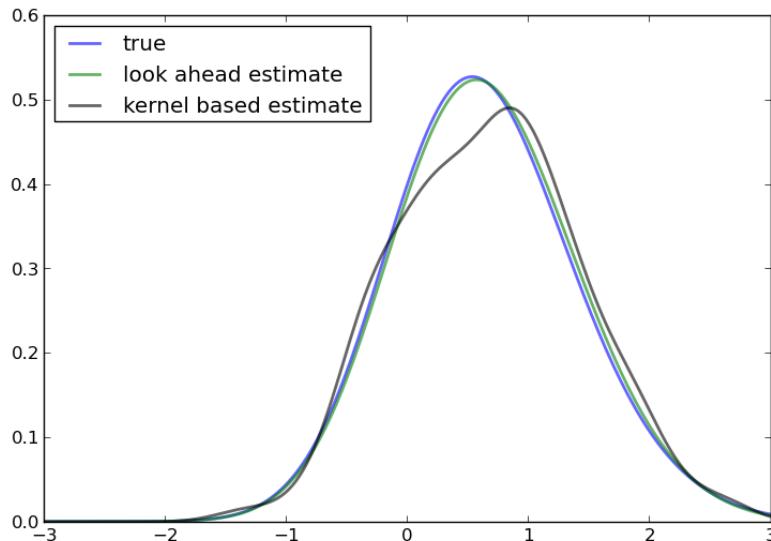
$$\psi^*(y) = 2 \phi(y) \Phi \left[ \frac{\theta y}{(1 - \theta^2)^{1/2}} \right] \quad (3.20)$$

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

As an exercise, compute the look ahead estimate of  $\psi^*$ , as defined in (3.18), and compare it with  $\psi^*$  in (3.20) to see whether they are indeed close for large  $n$

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

The next figure shows the result of such a computation



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

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

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

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

**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 [stationary\\_densities/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)
for ``i in range(4)``
```

**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

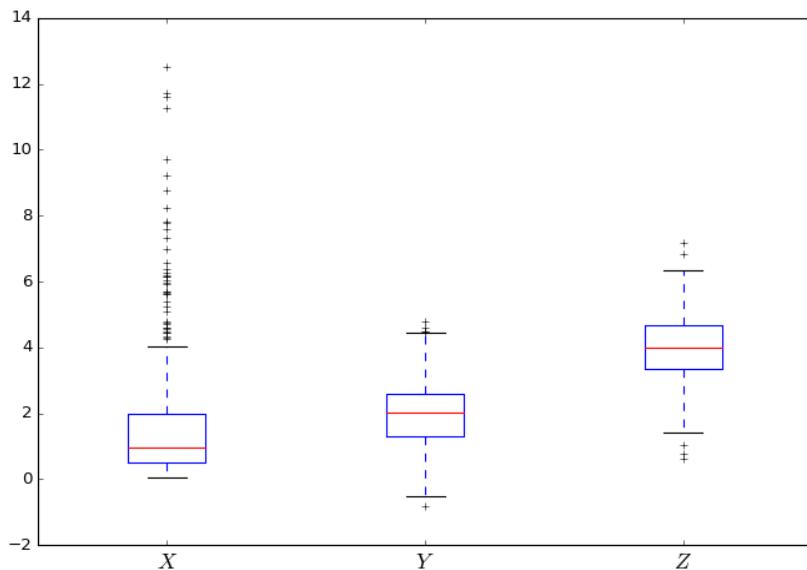
n = 500
x = np.random.randn(n)           #  $N(0, 1)$ 
x = np.exp(x)                   # Map x to lognormal
y = np.random.randn(n) + 2.0     #  $N(2, 1)$ 
z = np.random.randn(n) + 4.0     #  $N(4, 1)$ 

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

The three data sets are

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

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 (3.19)

We know that the distribution of  $X_t$  will converge to (3.20) whenever  $|\theta| < 1$

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

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

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

For each  $X_0$  in this set,

1. Generate  $k$  time series of length  $n$ , each starting at  $X_0$  and obeying (3.19)
2. Create a boxplot representing  $n$  distributions, where the  $t$ -th distribution shows the  $k$  observations of  $X_t$

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

## Solutions

[Solution notebook](#)

## Appendix

Here's the proof of (3.6)

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

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

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

Differentiating with respect to  $v$  yields (3.6)

## 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  $\phi$  and  $y_t \geq 0$

An asset is a claim on all or part of this endowment stream

The consumption goods  $\{y_t\}_{t=0}^{\infty}$  are nonstorable, so holding assets is the only way to transfer wealth into the future

For the purposes of intuition, it's common to think of the productive unit as a "tree" that produces fruit

Based on this idea, a "Lucas tree" is a claim on the consumption endowment

**Consumers** A representative consumer ranks consumption streams  $\{c_t\}$  according to the time separable utility functional

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \quad (3.21)$$

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  $\pi_t$  of the claim held in each period

Thus, the consumer problem is to maximize (3.21) subject to

$$c_t + \pi_{t+1} p_t \leq \pi_t y_t + \pi_t p_t$$

along with  $c_t \geq 0$  and  $0 \leq \pi_t \leq 1$  at each  $t$

The decision to hold share  $\pi_t$  is actually made at time  $t - 1$

But this value is inherited as a state variable at time  $t$ , which explains the choice of subscript

**The dynamic program** We can write the consumer problem as a dynamic programming problem

Our first observation is that prices depend on current information, and current information is really just the endowment process up until the current period

In fact the endowment process is Markovian, so that the only relevant information is the current state  $y \in \mathbb{R}_+$  (dropping the time subscript)

This leads us to guess an equilibrium where price is a function  $p$  of  $y$

Remarks on the solution method

- Since this is a competitive (read: price taking) model, the consumer will take this function  $p$  as given
- In this way we determine consumer behavior given  $p$  and then use equilibrium conditions to recover  $p$
- This is the standard way to solve competitive equilibrium models

Using the assumption that price is a given function  $p$  of  $y$ , we write the value function and constraint as

$$v(\pi, y) = \max_{c, \pi'} \left\{ u(c) + \beta \int v(\pi', G(y, z)) \phi(dz) \right\}$$

subject to

$$c + \pi' p(y) \leq \pi y + \pi p(y) \quad (3.22)$$

We can invoke the fact that utility is increasing to claim equality in (3.22) and hence eliminate the constraint, obtaining

$$v(\pi, y) = \max_{\pi'} \left\{ u[\pi(y + p(y)) - \pi' p(y)] + \beta \int v(\pi', G(y, z)) \phi(dz) \right\} \quad (3.23)$$

The solution to this dynamic programming problem is an optimal policy expressing either  $\pi'$  or  $c$  as a function of the state  $(\pi, y)$

- Each one determines the other, since  $c(\pi, y) = \pi(y + p(y)) - \pi'(\pi, y)p(y)$

**Next steps** What we need to do now is determine equilibrium prices

It seems that to obtain these, we will have to

1. Solve this two dimensional dynamic programming problem for the optimal policy
2. Impose equilibrium constraints
3. Solve out for the price function  $p(y)$  directly

However, as Lucas showed, there is a related but more straightforward way to do this

**Equilibrium constraints** Since the consumption good is not storable, in equilibrium we must have  $c_t = y_t$  for all  $t$

In addition, since there is one representative consumer (alternatively, since all consumers are identical), there should be no trade in equilibrium

In particular, the representative consumer owns the whole tree in every period, so  $\pi_t = 1$  for all  $t$   
 Prices must adjust to satisfy these two constraints

**The equilibrium price function** Now observe that the first order condition for (3.23) can be written as

$$u'(c)p(y) = \beta \int v'_1(\pi', G(y, z))\phi(dz)$$

where  $v'_1$  is the derivative of  $v$  with respect to its first argument

To obtain  $v'_1$  we can simply differentiate the right hand side of (3.23) with respect to  $\pi$ , yielding

$$v'_1(\pi, y) = u'(c)(y + p(y))$$

Next we impose the equilibrium constraints while combining the last two equations to get

$$p(y) = \beta \int \frac{u'[G(y, z)]}{u'(y)} [G(y, z) + p(G(y, z))]\phi(dz) \quad (3.24)$$

In sequential rather than functional notation, we can also write this as

$$p_t = \mathbb{E}_t \left[ \beta \frac{u'(c_{t+1})}{u'(c_t)} (c_{t+1} + p_{t+1}) \right] \quad (3.25)$$

This is the famous consumption-based asset pricing equation

Before discussing it further we want to solve out for prices

**Solving the Model** Equation (3.24) 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

$$f(y) := u'(y)p(y) \quad (3.26)$$

so that (3.24) becomes

$$f(y) = h(y) + \beta \int f[G(y, z)]\phi(dz) \quad (3.27)$$

Here  $h(y) := \beta \int u'[G(y, z)]G(y, z)\phi(dz)$  is a function that depends only on the primitives

Equation (3.27) is a functional equation in  $f$

The plan is to solve out for  $f$  and convert back to  $p$  via (3.26)

To solve (3.27) 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

$$(Tf)(y) = h(y) + \beta \int f[G(y, z)]\phi(dz) \quad (3.28)$$

The reason we do this is that a solution to (3.27) now corresponds to a function  $f^*$  satisfying  $(Tf^*)(y) = f^*(y)$  for all  $y$

In other words, a solution is a *fixed point* of  $T$

This means that we can use fixed point theory to obtain and compute the solution

**A little fixed point theory** Let  $cb\mathbb{R}_+$  be the set of continuous bounded functions  $f: \mathbb{R}_+ \rightarrow \mathbb{R}_+$

We now show that

1.  $T$  has exactly one fixed point  $f^*$  in  $cb\mathbb{R}_+$
2. For any  $f \in cb\mathbb{R}_+$ , the sequence  $T^k f$  converges uniformly to  $f^*$

(Note: If you find the mathematics heavy going you can take 1–2 as given and skip to the *next section*)

Recall the [Banach contraction mapping theorem](#)

It tells us that the previous statements will be true if we can find an  $\alpha < 1$  such that

$$\|Tf - Tg\| \leq \alpha \|f - g\|, \quad \forall f, g \in cb\mathbb{R}_+ \quad (3.29)$$

Here  $\|h\| := \sup_{x \in \mathbb{R}_+} |h(x)|$

To see that (3.29) 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 (3.29) 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](#) from the [QuantEcon.applications](#) repo

We repeat it here for convenience

```

"""
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 endowment 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 import interp
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  $\text{len}(f) = \text{len}(\text{grid})$ 

    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: 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,
                       tree,
                       Tf=Tf)

price = f * grid**gamma

return price

```

An example of usage is given in the docstring and repeated here

```

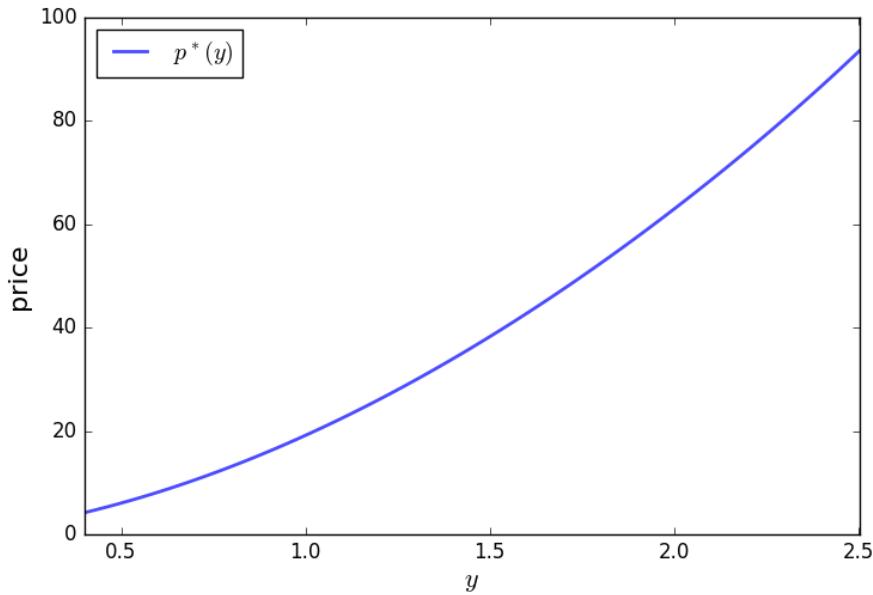
tree = LucasTree()
price_vals = compute_lt_price(tree)

```

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

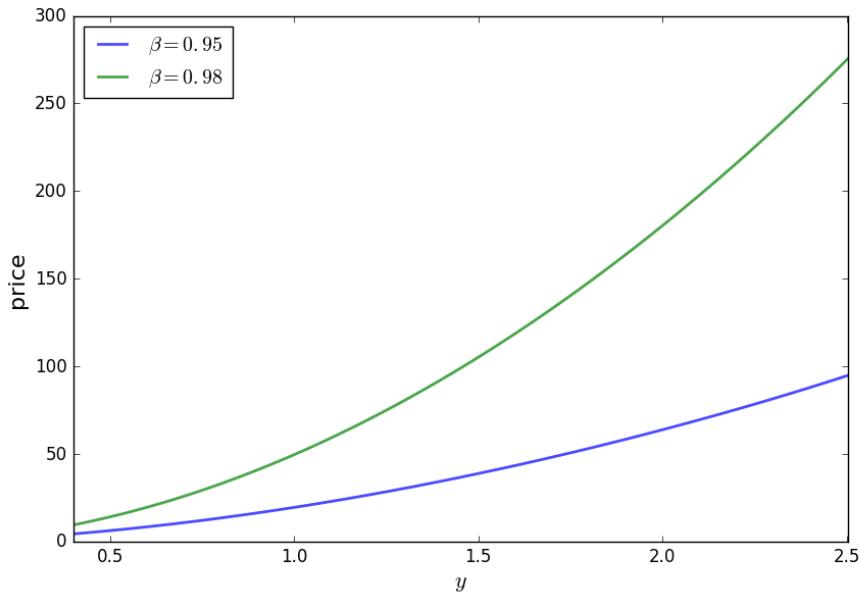
### Solutions

Solution notebook

## 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

$$a_{t+1} + c_t \leq wz_t + (1+r)a_t \quad c_t \geq 0, \quad \text{and} \quad a_t \geq -B$$

where

- $c_t$  is current consumption
- $a_t$  is assets
- $z_t$  is an exogenous component of labor income capturing stochastic unemployment risk, etc.
- $w$  is a wage rate
- $r$  is a net interest rate
- $B$  is the borrowing constraint

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  $\alpha$  are parameters with  $A > 0$  and  $\alpha \in (0, 1)$
- $K_t$  is aggregate capital
- $N$  is total labor supply (which is constant in this simple version of the model)

The firm's problem is

$$\max_{K,N} \left\{ AK_t^{\alpha}N^{1-\alpha} - rK - wN \right\}$$

From the first-order condition with respect to capital, the firm's inverse demand for capital is

$$r = A\alpha \left( \frac{N}{K} \right)^{1-\alpha} \tag{3.30}$$

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

$$w(r) = A(1 - \alpha)(\alpha / (1 + r))^{\alpha / (1 - \alpha)} \quad (3.31)$$

**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 (3.30) and a wage rate  $w(r)$  as given in (3.31)
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` from the `QuantEcon.applications` repository

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

As a first example of what we can do, let's plot an optimal accumulation policy at a given interest rate

```

"""
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 aiyagari_household import Household
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

```

```

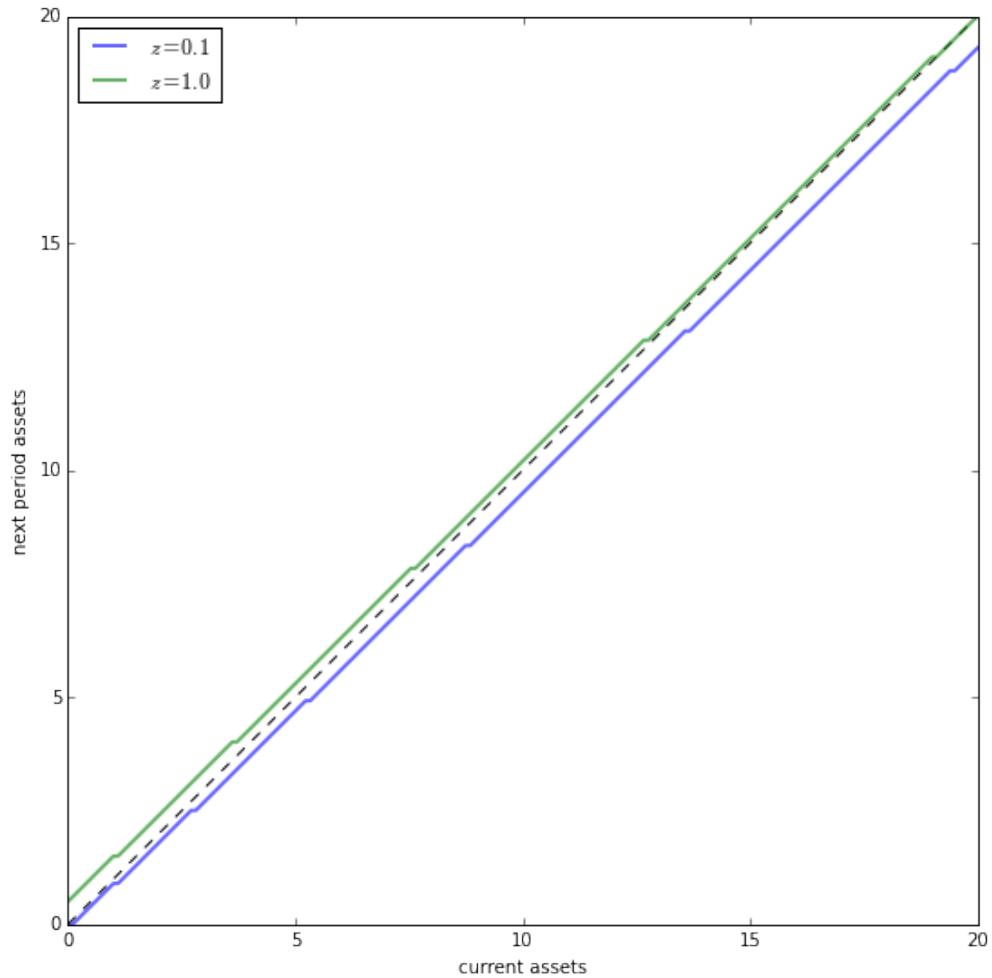
a_star = np.empty((z_size, a_size))
for s_i in range(n):
    a_i = s_i // z_size
    z_i = s_i % z_size
    a_star[z_i, a_i] = a_vals[results.sigma[s_i]]

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 aiyagari_household import Household, asset_marginal
from quantecon.markov import DiscreteDP

A = 2.5
N = 0.05
alpha = 0.33
beta = 0.96

def r_to_w(r):
    return A * (1 - alpha) * (alpha / (1 + r))**(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)

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()

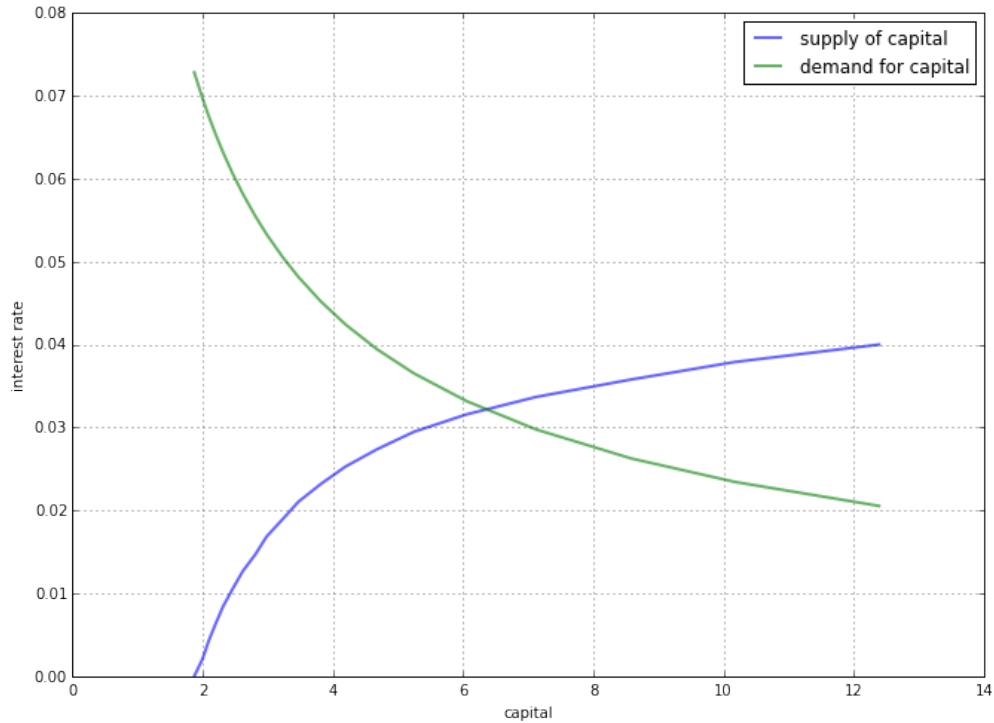
```

Here's the corresponding plot

## 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](#)] and 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
  - $\theta_t$  is contribution of career at time  $t$
  - $\epsilon_t$  is contribution of job at time  $t$

At the start of time  $t$ , a worker has the following options

- retain a current (career, job) pair  $(\theta_t, \epsilon_t)$  — referred to hereafter as “stay put”
- retain a current career  $\theta_t$  but redraw a job  $\epsilon_t$  — referred to hereafter as “new job”
- redraw both a career  $\theta_t$  and a job  $\epsilon_t$  — referred to hereafter as “new life”

Draws of  $\theta$  and  $\epsilon$  are independent of each other and past values, with

- $\theta_t \sim F$
- $\epsilon_t \sim G$

Notice that the worker does not have the option to retain a job but redraw a career — starting a new career always requires starting a new job

A young worker aims to maximize the expected sum of discounted wages

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t w_t \quad (3.32)$$

subject to the choice restrictions specified above

Let  $V(\theta, \epsilon)$  denote the value function, which is the maximum of (3.32) over all feasible (career, job) policies, given the initial state  $(\theta, \epsilon)$

The value function obeys

$$V(\theta, \epsilon) = \max\{I, II, III\},$$

where

$$\begin{aligned} I &= \theta + \epsilon + \beta V(\theta, \epsilon) \\ II &= \theta + \int \epsilon' G(d\epsilon') + \beta \int V(\theta, \epsilon') G(d\epsilon') \\ III &= \int \theta' F(d\theta') + \int \epsilon' G(d\epsilon') + \beta \int \int V(\theta', \epsilon') G(d\epsilon') F(d\theta') \end{aligned} \quad (3.33)$$

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  $\theta$  and  $\epsilon$  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

$$p(k | n, a, b) = \binom{n}{k} \frac{B(k+a, n-k+b)}{B(a, b)}, \quad k = 0, \dots, n$$

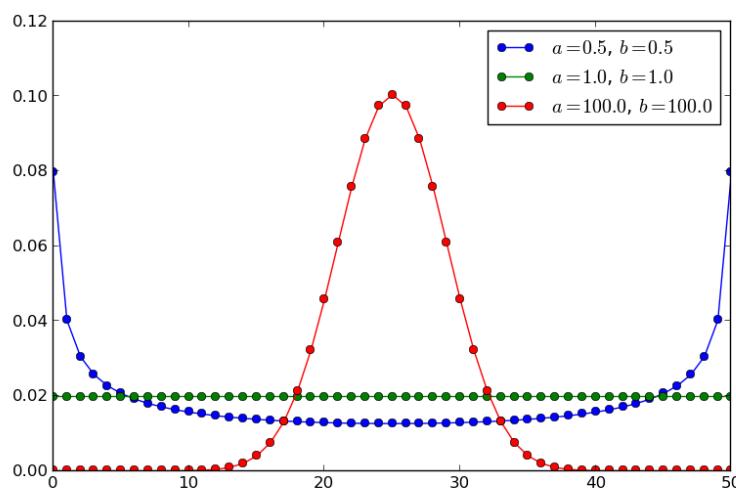
Interpretation:

- draw  $q$  from a Beta distribution with shape parameters  $(a, b)$
- run  $n$  independent binary trials, each with success probability  $q$
- $p(k | n, a, b)$  is the probability of  $k$  successes in these  $n$  trials

Nice properties:

- very flexible class of distributions, including uniform, symmetric unimodal, etc.
- only three parameters

Here's a figure showing the effect of different shape parameters when  $n = 50$



The code that generated this figure can be found [here](#)

**Implementation:** `career.py`

The QuantEcon.applications repo provides some code for solving the DP problem described above. See in particular [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.

```

"""
from textwrap import dedent
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 probabilities, discount factor and sample space for the variables.

    Parameters
    -----
    beta : scalar(float), optional(default=5.0)
        Discount factor
    B : scalar(float), optional(default=0.95)
        Upper bound of for both 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 __repr__(self):
    m = "CareerWorkerProblem(beta={b:g}, B={B:g}, N={n:g}, F_a={fa:g}, "
    m += "F_b={fb:g}, G_a={ga:g}, G_b={gb:g})"
    return m.format(b=self.beta, B=self.B, n=self.N, fa=self._F_a,
                    fb=self._F_b, ga=self._G_a, gb=self._G_b)

def __str__(self):
    m = """\
    CareerWorkerProblem (Neal, 1999)
        - beta (discount factor) : {b:g}
        - B (upper bound for epsilon and theta) : {B:g}
        - N (number of realizations of epsilon and theta) : {n:g}
        - F_a (parameter a from career distribution) : {fa:g}
        - F_b (parameter b from career distribution) : {fb:g}
        - G_a (parameter a from job distribution) : {ga:g}
        - G_b (parameter b from job distribution) : {gb:g}
    """
    return dedent(m.format(b=self.beta, B=self.B, n=self.N, fa=self._F_a,
                           fb=self._F_b, ga=self._G_a, gb=self._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 (3.33), 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

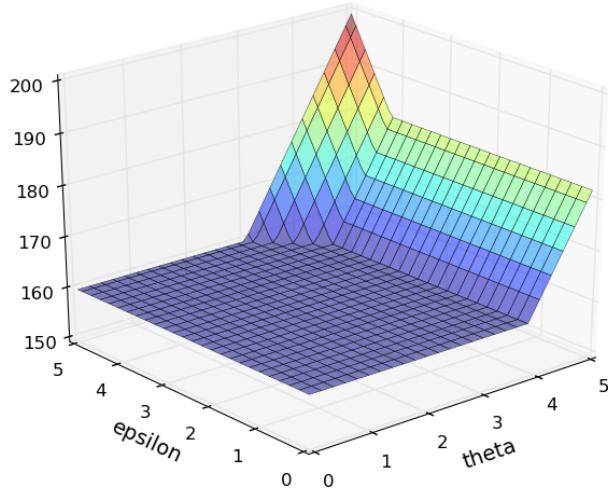


Fig. 3.1: Value function with uniform probabilities

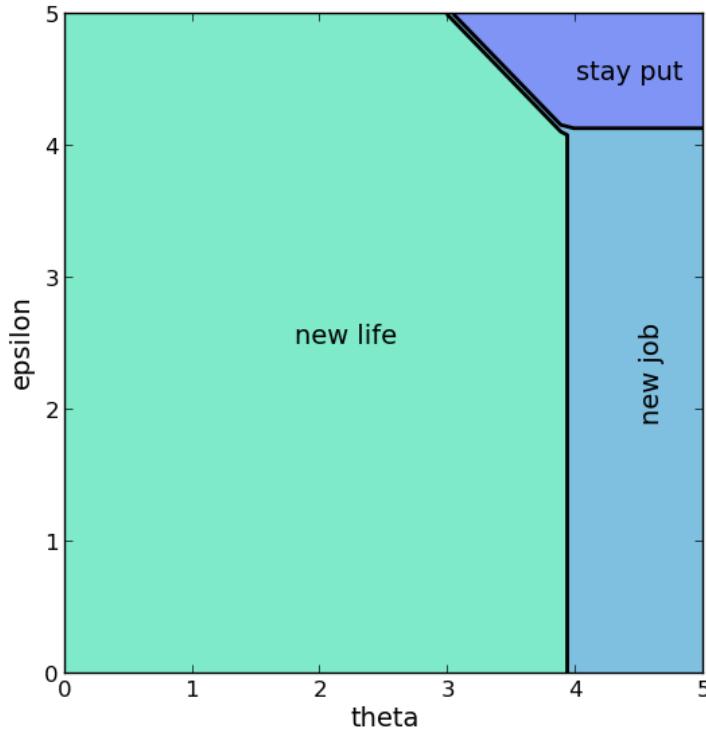
The code used to produce this plot was `career/career_vf_plot.py`

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  $\theta$  and  $\epsilon$  when the worker follows the optimal policy

In particular, modulo randomness, reproduce the following figure (where the horizontal axis represents time)

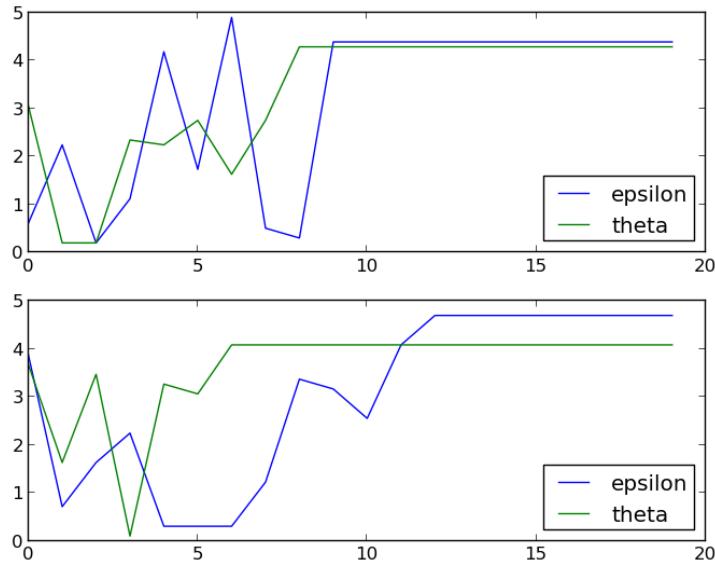
Hint: To generate the draws from the distributions  $F$  and  $G$ , use the class `DiscreteRV`

**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

$T^* :=$  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  $(\theta_t, \epsilon_t)$  enters the "stay put" region of  $(\theta, \epsilon)$  space



Letting  $S$  denote this region,  $T^*$  can be expressed as the first passage time to  $S$  under the optimal policy:

$$T^* := \inf\{t \geq 0 \mid (\theta_t, \epsilon_t) \in S\}$$

Collect 25,000 draws of this random variable and compute the median (which should be about 7)

Repeat the exercise with  $\beta = 0.99$  and interpret the change

**Exercise 3** As best you can, reproduce *the figure showing the optimal policy*

Hint: The `get_greedy()` 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

[Solution notebook](#)

## 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
- see also [add Jovanovic reference]

### 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

- $\phi_t$  is investment in job-specific human capital for the current role
- $s_t$  is search effort, devoted to obtaining new offers from other firms.

For as long as the worker remains in the current job, evolution of  $\{x_t\}$  is given by  $x_{t+1} = G(x_t, \phi_t)$

When search effort at  $t$  is  $s_t$ , the worker receives a new job offer with probability  $\pi(s_t) \in [0, 1]$

Value of offer is  $U_{t+1}$ , where  $\{U_t\}$  is iid with common distribution  $F$

Worker has the right to reject the current offer and continue with existing job.

In particular,  $x_{t+1} = U_{t+1}$  if accepts and  $x_{t+1} = G(x_t, \phi_t)$  if rejects

Letting  $b_{t+1} \in \{0, 1\}$  be binary with  $b_{t+1} = 1$  indicating an offer, we can write

$$x_{t+1} = (1 - b_{t+1})G(x_t, \phi_t) + b_{t+1} \max\{G(x_t, \phi_t), U_{t+1}\} \quad (3.34)$$

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 (3.34), the Bellman equation for this problem can be written as

$$V(x) = \max_{s+\phi \leq 1} \left\{ x(1-s-\phi) + \beta(1-\pi(s))V[G(x, \phi)] + \beta\pi(s) \int V[G(x, \phi) \vee u]F(du) \right\}. \quad (3.35)$$

Here nonnegativity of  $s$  and  $\phi$  is understood, while  $a \vee b := \max\{a, b\}$

**Parameterization** In the implementation below, we will focus on the parameterization

$$G(x, \phi) = A(x\phi)^\alpha, \quad \pi(s) = \sqrt{s} \quad \text{and} \quad F = \text{Beta}(2, 2)$$

with default parameter values

- $A = 1.4$
- $\alpha = 0.6$
- $\beta = 0.96$

The Beta(2,2) distribution is supported on  $(0, 1)$ . It has a unimodal, symmetric density peaked at 0.5.

**Back-of-the-Envelope Calculations** Before we solve the model, let's make some quick calculations that provide intuition on what the solution should look like.

To begin, observe that the worker has two instruments to build capital and hence wages:

1. invest in capital specific to the current job via  $\phi$
2. search for a new job with better job-specific capital match via  $s$

Since wages are  $x(1-s-\phi)$ , marginal cost of investment via either  $\phi$  or  $s$  is identical

Our risk neutral worker should focus on whatever instrument has the highest expected return

The relative expected return will depend on  $x$

For example, suppose first that  $x = 0.05$

- If  $s = 1$  and  $\phi = 0$ , then since  $G(x, \phi) = 0$ , taking expectations of (3.34) gives expected next period capital equal to  $\pi(s)\mathbb{E}U = \mathbb{E}U = 0.5$
- If  $s = 0$  and  $\phi = 1$ , then next period capital is  $G(x, \phi) = G(0.05, 1) \approx 0.23$

Both rates of return are good, but the return from search is better

Next suppose that  $x = 0.4$

- If  $s = 1$  and  $\phi = 0$ , then expected next period capital is again 0.5
- If  $s = 0$  and  $\phi = 1$ , then  $G(x, \phi) = G(0.4, 1) \approx 0.8$

Return from investment via  $\phi$  dominates expected return from search

Combining these observations gives us two informal predictions:

1. At any given state  $x$ , the two controls  $\phi$  and  $s$  will function primarily as substitutes — worker will focus on whichever instrument has the higher expected return
2. For sufficiently small  $x$ , search will be preferable to investment in job-specific human capital. For larger  $x$ , the reverse will be true

Now let's turn to implementation, and see if we can match our predictions.

### 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
from scipy import interp

# 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):
    """
    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
*  $\phi$  = investment in human capital
*  $\pi(s)$  = probability of new offer given search level  $s$ 
*  $x(1 - \phi - s)$  = wage
*  $G(x, \phi)$  = new human capital when current job retained
*  $U$  =  $RV$  with distribution  $F$  -- new draw of human capital

Parameters
-----
A : scalar(float), optional(default=1.4)
    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 capital
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: 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  $s + \phi \leq 1$ 
c2 = lambda z: z[0] - epsilon         # used to enforce  $s \geq \epsilon$ 
c3 = lambda z: z[1] - epsilon         # used to enforce  $\phi \geq \epsilon$ 
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

$$w(s, \phi) := - \left\{ x(1 - s - \phi) + \beta(1 - \pi(s))V[G(x, \phi)] + \beta\pi(s) \int V[G(x, \phi) \vee u]F(du) \right\} \quad (3.36)$$

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 (3.36), 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  $\phi$

Inside the `for` loop, for each `x` in the grid over the state space, we set up the function  $w(z) = w(s, \phi)$  defined in (3.36).

The function is minimized over all feasible  $(s, \phi)$  pairs, either by

- a relatively sophisticated solver from SciPy called `fmin_slsqp`, or
- brute force search over a grid

The former is much faster, but convergence to the global optimum is not guaranteed. Grid search is a simple way to check results

### Solving for Policies

Let's plot the optimal policies and see what they look like

The code is in a file `jv/jv_test.py` from the [applications repository](#) and looks as follows

```
import matplotlib.pyplot as plt
from quantecon import compute_fixed_point
from jv import JvWorker

# === 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

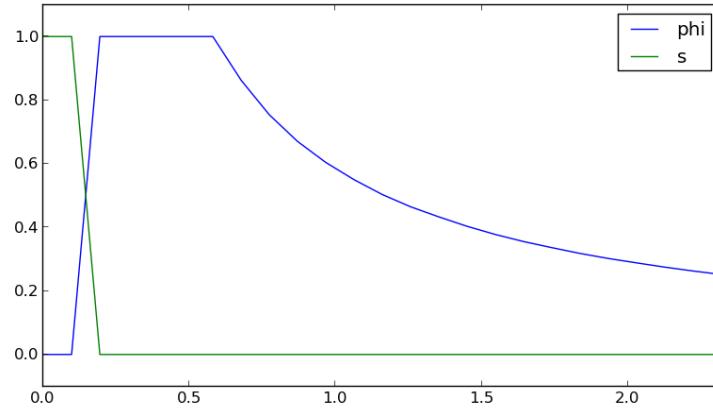


Fig. 3.2: Optimal policies

The horizontal axis is the state  $x$ , while the vertical axis gives  $s(x)$  and  $\phi(x)$

Overall, the policies match well with our predictions from section.

- Worker switches from one investment strategy to the other depending on relative return
- For low values of  $x$ , the best option is to search for a new job

- Once  $x$  is larger, worker does better by investing in human capital specific to the current position

### Exercises

**Exercise 1** Let's look at the dynamics for the state process  $\{x_t\}$  associated with these policies.

The dynamics are given by (3.34) when  $\phi_t$  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  $\phi_t$  converges to about 0.6

Since these results were calculated at a value of  $\beta$  close to one, let's compare them to the best choice for an *infinitely* patient worker.

Intuitively, an infinitely patient worker would like to maximize steady state wages, which are a function of steady state capital.

You can take it as given—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  $\phi$ , steady state capital is the positive fixed point  $x^*(\phi)$  of the map  $x \mapsto G(x, \phi)$ .

Steady state wages can be written as  $w^*(\phi) = x^*(\phi)(1 - \phi)$

Graph  $w^*(\phi)$  with respect to  $\phi$ , and examine the best choice of  $\phi$

Can you give a rough interpretation for the value that you see?

### Solutions

[Solution notebook](#)

## 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

$$V(w) = \max \left\{ \frac{w}{1-\beta}, c + \beta \int V(w') h(w') dw' \right\} \quad (3.37)$$

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  $(\beta, h, c)$  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  $\pi_0$  on  $f$  being chosen

Update rule: worker’s time  $t$  estimate of the distribution is  $\pi_t f + (1 - \pi_t) g$ , where  $\pi_t$  updates via

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t) g(w_{t+1})} \quad (3.38)$$

This last expression follows from Bayes’ rule, which tells us that

$$\mathbb{P}\{h = f \mid W = w\} = \frac{\mathbb{P}\{W = w \mid h = f\} \mathbb{P}\{h = f\}}{\mathbb{P}\{W = w\}} \quad \text{and} \quad \mathbb{P}\{W = w\} = \sum_{\psi \in \{f,g\}} \mathbb{P}\{W = w \mid h = \psi\} \mathbb{P}\{h = \psi\}$$

The fact that (3.38) 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

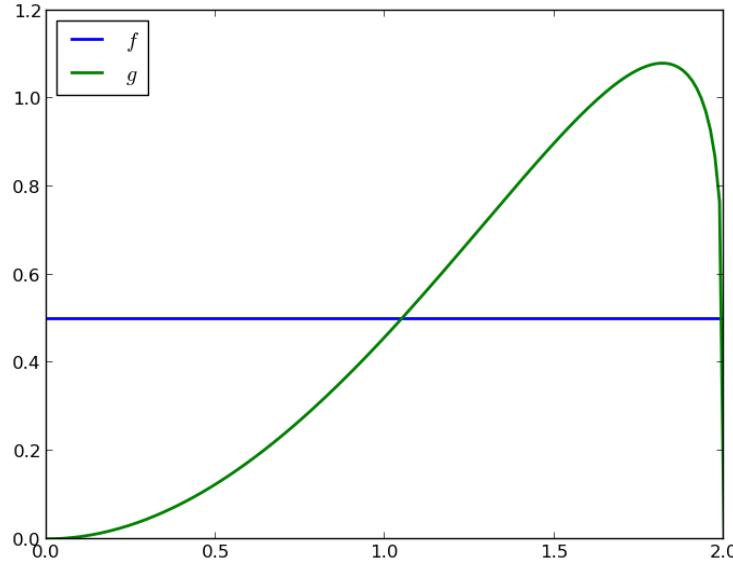
$$V(w, \pi) = \max \left\{ \frac{w}{1 - \beta}, c + \beta \int V(w', \pi') h_\pi(w') dw' \right\} \quad \text{where} \quad \pi' = q(w', \pi) \quad (3.39)$$

Notice that the current guess  $\pi$  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 Beta(1,1) scaled (i.e., draws are multiplied by) some factor  $w_m$
- $g$  is Beta(3,1.2) scaled (i.e., draws are multiplied by) the same factor  $w_m$
- $\beta = 0.95$  and  $c = 0.6$

With  $w_m = 2$ , the densities  $f$  and  $g$  have the following shape



**Looking Forward** What kind of optimal policy might result from (3.39) and the parameterization specified above?

Intuitively, if we accept at  $w_a$  and  $w_a \leq w_b$ , then — all other things being given — we should also accept at  $w_b$

This suggests a policy of accepting whenever  $w$  exceeds some threshold value  $\bar{w}$

But  $\bar{w}$  should depend on  $\pi$  — in fact it should be decreasing in  $\pi$  because

- $f$  is a less attractive offer distribution than  $g$
  - larger  $\pi$  means more weight on  $f$  and less on  $g$

Thus larger  $\pi$  depresses the 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` contained in the `QuantEcon.applications` repository.

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 textwrap import dedent
from scipy.interpolate import LinearNDInterpolator
from scipy.integrate import fixed_quad
from scipy.stats import beta as beta_distribution
from scipy import interp
from numpy import maximum as npmax
import numpy as np

class SearchProblem(object):
    """
    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 __repr__(self):
    m = "SearchProblem(beta={b}, c={c}, F_a={fa}, F_b={fb}, G_a={ga}, "
    m += "G_b={gb}, w_max={wu}, w_grid_size={wgs}, pi_grid_size={pgs})"
    fa, fb = self.F.args
    ga, gb = self.G.args

```

```

        return m.format(b=self.beta, c=self.c, fa=fa, fb=fb, ga=ga,
                         gb=gb, wu=self.w_grid.max(),
                         wgs=self.w_grid.size, pgs=self.pi_grid.size)

    def __str__(self):
        m = """\
SearchProblem (offer distribution unknown):
- beta (discount factor) : {b:g}
- c (unemployment compensation) : {c}
- F (distribution F) : Beta({fa}, {fb:g})
- G (distribution G) : Beta({ga}, {gb:g})
- w bounds (bounds for wage offers) : ({wl:g}, {wu:g})
- w grid size (number of points in grid for wage) : {wgs}
- pi bounds (bounds for probability of dist f) : ({pl:g}, {pu:g})
- pi grid size (number of points in grid for pi) : {pgs}
"""

        fa, fb = self.F.args
        ga, gb = self.G.args
        return dedent(m.format(b=self.beta, c=self.c, fa=fa, fb=fb, ga=ga,
                               gb=gb,
                               wl=self.w_grid.min(), wu=self.w_grid.max(),
                               wgs=self.w_grid.size,
                               pl=self.pi_grid.min(), pu=self.pi_grid.max(),
                               pgs=self.pi_grid.size))

    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: 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

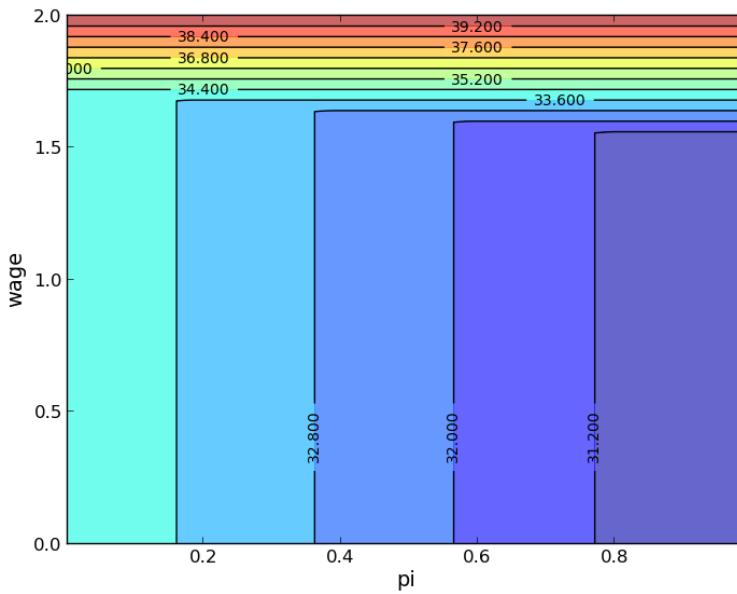
The Bellman operator is implemented as the method `bellman_operator()`, while `get_greedy()` computes an approximate optimal policy from a guess `v` of the value function

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 in file `odu/odu_vfi_plots.py` from the [applications repository](#)

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,

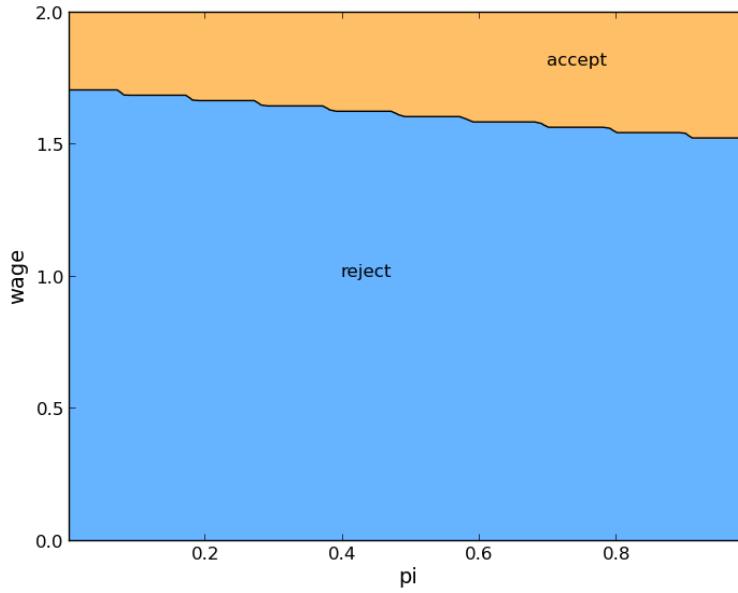
But instead of pursuing that, let's consider another method to solve for the optimal policy

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 (3.39) have equal value:

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int V(w', \pi') h_\pi(w') dw' \quad (3.40)$$

Together, (3.39) and (3.40) give

$$V(w, \pi) = \max \left\{ \frac{w}{1-\beta}, \frac{\bar{w}(\pi)}{1-\beta} \right\} \quad (3.41)$$

Combining (3.40) and (3.41), 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

$$\bar{w}(\pi) = (1 - \beta)c + \beta \int \max \{ w', \bar{w} \circ q(w', \pi) \} h_\pi(w') dw' \quad (3.42)$$

Equation (3.42) 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

$$(Q\psi)(\pi) = (1 - \beta)c + \beta \int \max \{w', \psi \circ q(w', \pi)\} h_\pi(w') dw' \quad (3.43)$$

Comparing (3.42) and (3.43), 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 (3.42) and vice versa

Moreover, for any  $\psi, \phi \in b[0, 1]$ , basic algebra and the triangle inequality for integrals tells us that

$$|(Q\psi)(\pi) - (Q\phi)(\pi)| \leq \beta \int |\max \{w', \psi \circ q(w', \pi)\} - \max \{w', \phi \circ q(w', \pi)\}| h_\pi(w') dw' \quad (3.44)$$

Working case by case, it is easy to check that for real numbers  $a, b, c$  we always have

$$|\max\{a, b\} - \max\{a, c\}| \leq |b - c| \quad (3.45)$$

Combining (3.44) and (3.45) yields

$$|(Q\psi)(\pi) - (Q\phi)(\pi)| \leq \beta \int |\psi \circ q(w', \pi) - \phi \circ q(w', \pi)| h_\pi(w') dw' \leq \beta \|\psi - \phi\| \quad (3.46)$$

Taking the supremum over  $\pi$  now gives us

$$\|Q\psi - Q\phi\| \leq \beta \|\psi - \phi\| \quad (3.47)$$

In other words,  $Q$  is a contraction of modulus  $\beta$  on the complete metric space  $(b[0, 1], \|\cdot\|)$

Hence

- A unique solution  $\bar{w}$  to the RWFE exists in  $b[0, 1]$
- $Q^k \psi \rightarrow \bar{w}$  uniformly as  $k \rightarrow \infty$ , 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

[Solution notebook](#)

# Optimal Savings

## Contents

- *Optimal Savings*
  - [Overview](#)
  - [The Optimal Savings Problem](#)
  - [Computation](#)
  - [Exercises](#)
  - [Solutions](#)

## Overview

Next we study the standard optimal savings problem for an infinitely lived consumer—the “common ancestor” described in [\[LS12\]](#), section 1.3

- Also known as the income fluctuation problem
- An important sub-problem for many representative macroeconomic models
  - [\[Aiy94\]](#)
  - [\[Hug93\]](#)
  - etc.
- Useful references include [\[Dea91\]](#), [\[DH10\]](#), [\[Kuh13\]](#), [\[Rab02\]](#), [\[Rei09\]](#) and [\[SE77\]](#)

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](#)

In this lecture we will explore an alternative to value function iteration (VFI) called *policy function iteration* (PFI)

- Based on the Euler equation, and not to be confused with Howard’s policy iteration algorithm
- Globally convergent under mild assumptions, even when utility is unbounded (both above and below)
- Numerically, turns out to be faster and more efficient than VFI for this model

### Model features

- Infinite horizon dynamic programming with two states and one control

#### The Optimal Savings Problem

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

$$c_t + a_{t+1} \leq Ra_t + z_t, \quad c_t \geq 0, \quad a_t \geq -b \quad t = 0, 1, \dots \quad (3.48)$$

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

#### Assumptions

1.  $\{z_t\}$  is a finite Markov process with Markov matrix  $\Pi$  taking values in  $Z$
2.  $|Z| < \infty$  and  $Z \subset (0, \infty)$
3.  $r > 0$  and  $\beta R < 1$
4.  $u$  is smooth, strictly increasing and strictly concave with  $\lim_{c \rightarrow 0} u'(c) = \infty$  and  $\lim_{c \rightarrow \infty} u'(c) = 0$

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 (3.48), and
3. measurability of  $c_t$  w.r.t. the filtration generated by  $\{z_1, \dots, z_t\}$

The meaning of the third point is just that consumption at time  $t$  can only be a function of outcomes that have already been observed

The *value function*  $V: S \rightarrow \mathbb{R}$  is defined by

$$V(a, z) := \sup \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\} \quad (3.49)$$

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 (3.49)

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

$$u'(c_t) = \max \{ \beta R \mathbb{E}_t[u'(c_{t+1})], u'(Ra_t + z_t + b) \} \quad (3.50)$$

and the transversality condition

$$\lim_{t \rightarrow \infty} \beta^t \mathbb{E}[u'(c_t)a_{t+1}] = 0. \quad (3.51)$$

Moreover, there exists an *optimal consumption function*  $c^*: S \rightarrow [0, \infty)$  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 (3.50) and (3.51), 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. Value function iteration (VFI)
2. Policy function iteration (PFI) using the Euler equality

### Policy function iteration

We can rewrite (3.50) to make it a statement about functions rather than random variables

In particular, consider the functional equation

$$u' \circ c(a, z) = \max \left\{ \gamma \int u' \circ c \{ Ra + z - c(a, z), \dot{z} \} \Pi(z, d\dot{z}), u'(Ra + z + b) \right\} \quad (3.52)$$

where  $\gamma := \beta R$  and  $u' \circ c(s) := u'(c(s))$

Equation (3.52) is a functional equation in  $c$

In order to identify a solution, let  $\mathcal{C}$  be the set of candidate consumption functions  $c: S \rightarrow \mathbb{R}$  such that

- each  $c \in \mathcal{C}$  is continuous and (weakly) increasing
- $\min Z \leq c(a, z) \leq Ra + z + b$  for all  $(a, z) \in S$

In addition, let  $K: \mathcal{C} \rightarrow \mathcal{C}$  be defined as follows:

For given  $c \in \mathcal{C}$ , the value  $Kc(a, z)$  is the unique  $t \in J(a, z)$  that solves

$$u'(t) = \max \left\{ \gamma \int u' \circ c \{ Ra + z - t, \dot{z} \} \Pi(z, d\dot{z}), u'(Ra + z + b) \right\} \quad (3.53)$$

where

$$J(a, z) := \{t \in \mathbb{R} : \min Z \leq t \leq Ra + z + b\} \quad (3.54)$$

We refer to  $K$  as Coleman's policy function operator [Col90]

It is known that

- $K$  is a contraction mapping on  $\mathcal{C}$  under the metric

$$\rho(c, d) := \|u' \circ c - u' \circ d\| := \sup_{s \in S} |u'(c(s)) - u'(d(s))| \quad (c, d \in \mathcal{C})$$

- The metric  $\rho$  is complete on  $\mathcal{C}$
- Convergence in  $\rho$  implies uniform convergence on compacts

In consequence,  $K$  has a unique fixed point  $c^* \in \mathcal{C}$  and  $K^n c \rightarrow c^*$  as  $n \rightarrow \infty$  for any  $c \in \mathcal{C}$

By the definition of  $K$ , the fixed points of  $K$  in  $\mathcal{C}$  coincide with the solutions to (3.52) in  $\mathcal{C}$

In particular, it can be shown that the path  $\{c_t\}$  generated from  $(a_0, z_0) \in S$  using policy function  $c^*$  is the unique optimal path from  $(a_0, z_0) \in S$

**TL;DR** The unique optimal policy can be computed by picking any  $c \in \mathcal{C}$  and iterating with the operator  $K$  defined in (3.53)

### Value function iteration

The Bellman operator for this problem is given by

$$Tv(a, z) = \max_{0 \leq c \leq Ra + z + b} \left\{ u(c) + \beta \int v(Ra + z - c, \dot{z}) \Pi(z, d\dot{z}) \right\} \quad (3.55)$$

We have to be careful with VFI (i.e., iterating with  $T$ ) in this setting because  $u$  is not assumed to be bounded

- In fact typically unbounded both above and below — e.g.  $u(c) = \log c$
- In which case, the standard DP theory does not apply
- $T^n v$  is not guaranteed to converge to the value function for arbitrary continuous bounded  $v$

Nonetheless, we can always try the strategy "iterate and hope"

- In this case we can check the outcome by comparing with PFI
- The latter is known to converge, as described above

**Implementation** The code in `ifp.py` from `QuantEcon.applications` provides implementations of both VFI and PFI

The code is repeated here and a description and clarifications are given below

```
"""
Filename: ifp.py
```

Authors: Thomas Sargent, John Stachurski

*Tools for solving the standard optimal savings / income fluctuation problem for an infinitely lived consumer facing an exogenous income process that evolves according to a Markov chain.*

#### References

-----

<http://quant-econ.net/py/ifp.html>

```
"""
from textwrap import dedent
import numpy as np
from scipy.optimize import fminbound, brentq
from scipy import interp

class ConsumerProblem(object):
    """
    A class for solving the income fluctuation problem. Iteration with
    either the Coleman or Bellman operators from appropriate initial
    conditions leads to convergence to the optimal consumption policy.
    The income process is a finite state Markov chain. Note that the
    Coleman operator is the preferred method, as it is almost always
    faster and more accurate. The Bellman operator is only provided for
    comparison.

```

#### 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 __repr__(self):
    m = "ConsumerProblem(r={r:g}, beta={be:g}, Pi='{n:g} by {n:g}', "
    m += "z_vals={z}, b={b:g}, grid_max={gm:g}, grid_size={gs:g}, "
    m += "u={u}, du={du})"
    return m.format(r=self.r, be=self.beta, n=self.Pi.shape[0],
                   z=self.z_vals, b=self.b,
                   gm=self.asset_grid.max(), gs=self.asset_grid.size,
                   u=self.u, du=self.du)

def __str__(self):
    m = """
    Consumer Problem (optimal savings):
    - r (interest rate) : {r:g}
    - beta (discount rate) : {be:g}
    - Pi (transition matrix) : {n} by {n}
    - z_vals (state space of shocks) : {z}
    - b (borrowing constraint) : {b:g}
    - grid_max (maximum of asset grid) : {gm:g}
    - grid_size (number of points in asset grid) : {gs:g}
    - u (utility function) : {u}
    - du (marginal utility function) : {du}
    """
    return dedent(m.format(r=self.r, be=self.beta, n=self.Pi.shape[0],
                           z=self.z_vals, b=self.b,
                           gm=self.asset_grid.max(),
                           gs=self.asset_grid.size, u=self.u,
                           du=self.du))

def bellman_operator(self, V, 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)
    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 = self.R, self.Pi, self.beta, self.u, self.b
asset_grid, z_vals = self.asset_grid, self.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: 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, np.min(z_vals), 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(self, c):
    """
The approximate Coleman operator.

Iteration with this operator corresponds to policy function
iteration. 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)

>Returns
-----
array_like(float)
    The updated policy, where updating is by the Coleman
    operator. function  $TV$ .

"""

```

```

# === simplify names, set up arrays === #
R, Pi, beta, du, b = self.R, self.Pi, self.beta, self.du, self.b
asset_grid, z_vals = self.asset_grid, self.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] = 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, np.min(z_vals), R * a + z + b)

return Kc

def initialize(self):
    """
    Creates a suitable initial conditions V and c for value function and policy function iteration respectively.

    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 = self.R, self.beta, self.u, self.b
    asset_grid, z_vals = self.asset_grid, self.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

```

The code contains a class called `ConsumerProblem` that

- stores all the relevant parameters of a given model
- defines methods
  - `bellman_operator`, which implements the Bellman operator  $T$  specified above
  - `coleman_operator`, which implements the Coleman operator  $K$  specified above
  - `initialize`, which generates suitable initial conditions for iteration

The methods `bellman_operator` and `coleman_operator` both 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 PFI produce similar results, the latter is much faster

- Because we are exploiting analytically derived first order conditions

Another benefit of working in policy function space rather than value function space is that value functions typically have more curvature

- Makes them harder to approximate numerically

## Exercises

**Exercise 1** The first exercise is to replicate the following figure, which compares PFI 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

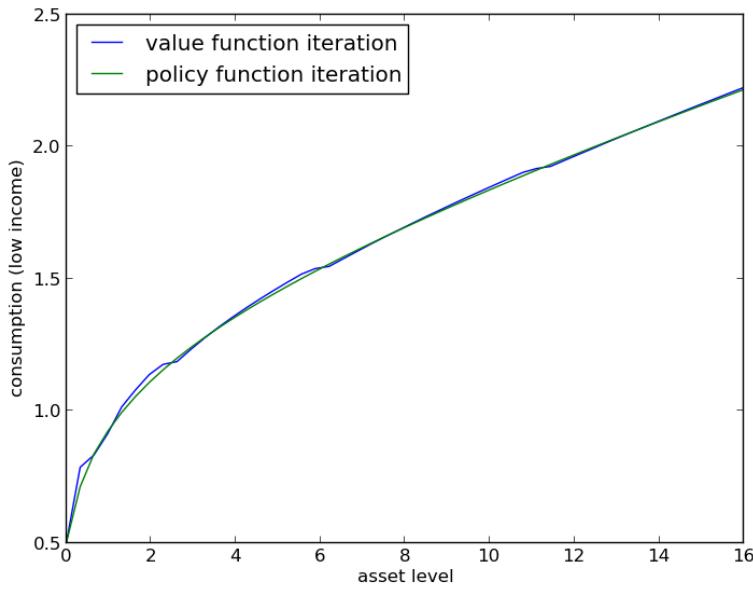
Consumption is shown as a function of assets with income  $z$  held fixed at its smallest value

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



```
In [1]: run ifp.py
```

```
In [2]: import quantecon as qe
```

```
In [3]: cp = ConsumerProblem()
```

```
In [4]: v, c = cp.initialize()
```

```
In [5]: timeit cp.bellman_operator(v)
10 loops, best of 3: 142 ms per loop
```

```
In [6]: timeit cp.coleman_operator(c)
10 loops, best of 3: 24.9 ms per loop
```

The output shows that Coleman operator is about 6 times faster

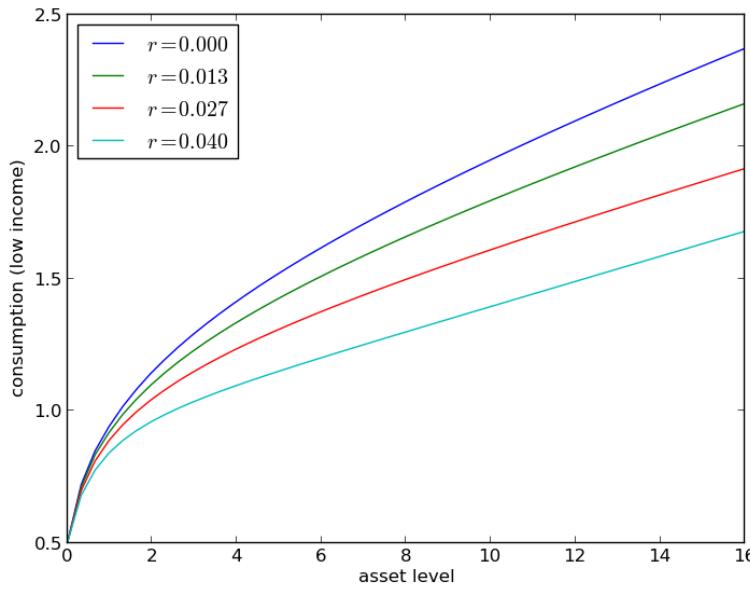
From now on we will only use the Coleman operator

**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 respectively

The dashed line is the 45 degree line

We can see from the figure that the dynamics will be stable — assets do not diverge

In fact there is a unique stationary distribution of assets that we can calculate by simulation

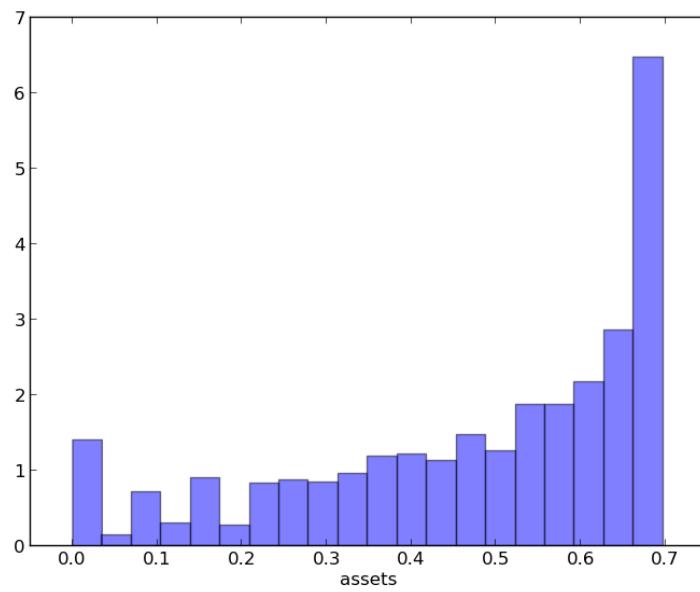
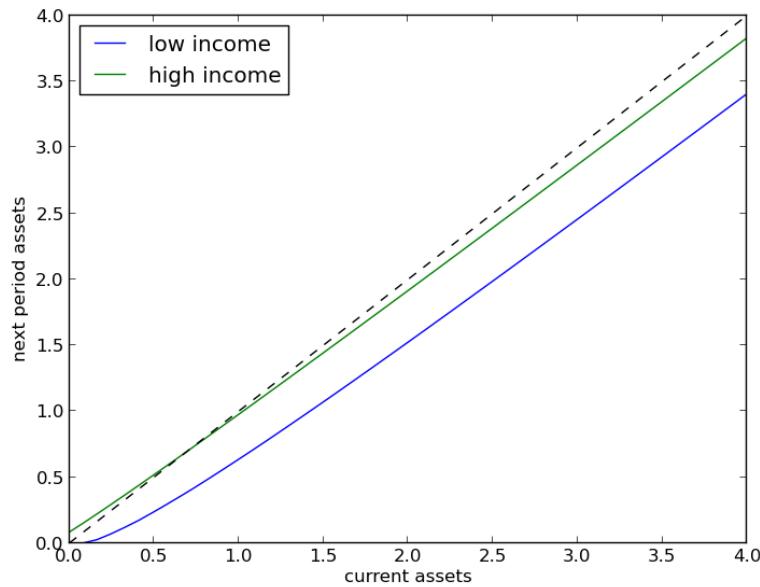
- Can be proved via theorem 2 of [HP92]
- Represents the long run dispersion of assets across households when households have idiosyncratic shocks

Ergodicity is valid here, so stationary probabilities can be calculated by averaging over a single long time series

- Hence to approximate the stationary distribution we can simulate a long time series for assets and histogram, as in the following figure

Your task is to replicate the figure

- Parameters are as discussed above
- The histogram in the figure used a single time series  $\{a_t\}$  of length 500,000



- Given the length of this time series, the initial condition  $(a_0, z_0)$  will not matter
- You might find it helpful to use the `MarkovChain` class from `quantecon`

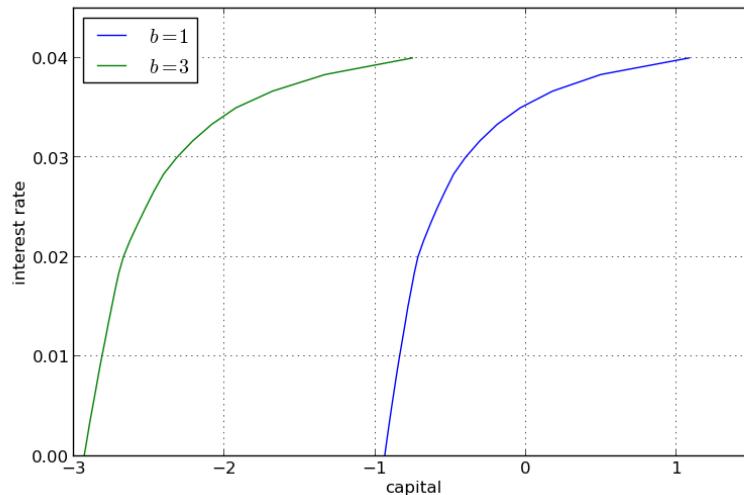
**Exercise 4** Following on from exercises 2 and 3, let's look at how savings and aggregate asset holdings vary with the interest rate

- Note: [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

[Solution notebook](#)

## 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 advantage of being

1. simple enough to be described by an elegant and comprehensive theory
2. relatively broad in terms of the kinds of dynamics it can represent

We consider these models in both the time and frequency domain

**ARMA Processes** We will focus much of our attention on linear covariance stationary models with a finite number of parameters

In particular, we will study stationary ARMA processes, which form a cornerstone of the standard theory of time series analysis

It's well known that 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 of a covariance stationary process

Having a second representation of this important object

- shines new light on the dynamics of the process in question
- allows for a simpler, more tractable representation in certain 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 requires identifying some deep structure in this process that is relatively constant over time

If such structure can be found, then each new observation  $X_t, X_{t+1}, \dots$  provides additional information about it — which is how we learn from data

For this reason, we will focus in what follows on processes that are *stationary* — or become so after some transformation (differencing, cointegration, etc.)

**Definitions** A real-valued stochastic process  $\{X_t\}$  is called *covariance stationary* if

1. Its mean  $\mu := \mathbb{E}X_t$  does not depend on  $t$
2. For all  $k$  in  $\mathbb{Z}$ , the  $k$ -th autocovariance  $\gamma(k) := \mathbb{E}(X_t - \mu)(X_{t+k} - \mu)$  is finite and depends only on  $k$

The function  $\gamma: \mathbb{Z} \rightarrow \mathbb{R}$  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)

**Example 2: General Linear Processes** From the simple building block provided by white noise, we can construct a very flexible family of covariance stationary processes — the *general linear processes*

$$X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}, \quad t \in \mathbb{Z} \quad (3.56)$$

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*

With some manipulations it is possible to confirm that the autocovariance function for (3.56) is

$$\gamma(k) = \sigma^2 \sum_{j=0}^{\infty} \psi_j \psi_{j+k} \quad (3.57)$$

By the *Cauchy-Schwartz inequality* one can show that the last expression is finite. Clearly it 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 theorem* states that every zero-mean covariance stationary process  $\{X_t\}$  can be written as

$$X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j} + \eta_t$$

where

- $\{\epsilon_t\}$  is white noise
- $\{\psi_t\}$  is square summable
- $\eta_t$  can be expressed as a linear function of  $X_{t-1}, X_{t-2}, \dots$  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, and it often pays to specialize to those for which there exists a representation having only finitely many parameters

(In fact, experience shows 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 AR(1) process

$$X_t = \phi X_{t-1} + \epsilon_t \quad \text{where } |\phi| < 1 \quad \text{and } \{\epsilon_t\} \text{ is white noise} \quad (3.58)$$

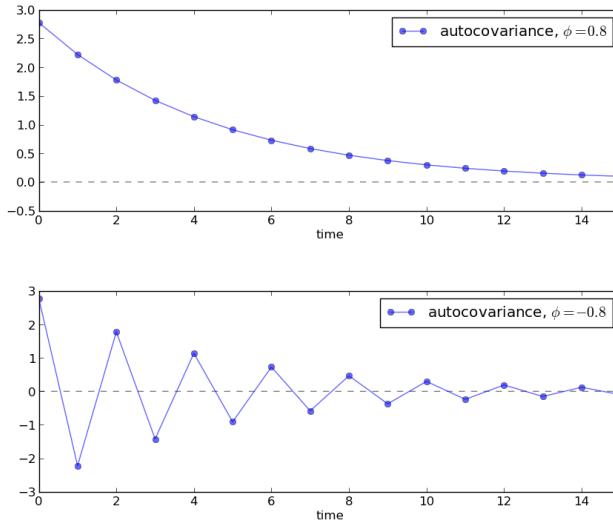
By direct substitution, it is easy to verify that  $X_t = \sum_{j=0}^{\infty} \phi^j \epsilon_{t-j}$

Hence  $\{X_t\}$  is a general linear process

Applying (3.57) to the previous expression for  $X_t$ , we get the AR(1) autocovariance function

$$\gamma(k) = \phi^k \frac{\sigma^2}{1 - \phi^2}, \quad k = 0, 1, \dots \quad (3.59)$$

The next figure plots this function for  $\phi = 0.8$  and  $\phi = -0.8$  with  $\sigma = 1$



Another very simple process is the MA(1) process

$$X_t = \epsilon_t + \theta \epsilon_{t-1}$$

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

$$X_t = \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q} \quad (3.60)$$

where  $\{\epsilon_t\}$  is white noise

There is an alternative notation for ARMA processes in common use, based around the *lag operator*  $L$

**Def.** Given arbitrary variable  $Y_t$ , let  $L^k Y_t := Y_{t-k}$

It turns out that

- lag operators can lead to very succinct expressions for linear stochastic processes

- algebraic manipulations treating the lag operator as an ordinary scalar often are legitimate

Using  $L$ , we can rewrite (3.60) as

$$L^0 X_t - \phi_1 L^1 X_t - \cdots - \phi_p L^p X_t = L^0 \epsilon_t + \theta_1 L^1 \epsilon_t + \cdots + \theta_q L^q \epsilon_t \quad (3.61)$$

If we let  $\phi(z)$  and  $\theta(z)$  be the polynomials

$$\phi(z) := 1 - \phi_1 z - \cdots - \phi_p z^p \quad \text{and} \quad \theta(z) := 1 + \theta_1 z + \cdots + \theta_q z^q \quad (3.62)$$

then (3.61) simplifies further to

$$\phi(L) X_t = \theta(L) \epsilon_t \quad (3.63)$$

In what follows we **always assume** that the roots of the polynomial  $\phi(z)$  lie outside the unit circle in the complex plane

This condition is sufficient to guarantee that the ARMA( $p, q$ ) process is covariance stationary

In fact it implies that the process falls within the class of general linear processes *described above*

That is, given an ARMA( $p, q$ ) process  $\{X_t\}$  satisfying the unit circle condition, there exists a square summable sequence  $\{\psi_t\}$  with  $X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}$  for all  $t$

The sequence  $\{\psi_t\}$  can be obtained by a recursive procedure outlined on page 79 of [CC08]

In this context, the function  $t \mapsto \psi_t$  is often called the *impulse response function*

## Spectral Analysis

Autocovariance functions provide a great deal of information about covariance stationary processes

In fact, for zero-mean Gaussian processes, the autocovariance function characterizes the entire joint distribution

Even for non-Gaussian processes, it provides a significant amount of information

It turns out that there is an alternative representation of the autocovariance function of a covariance stationary process, called the *spectral density*

At times, the spectral density is easier to derive, easier to manipulate and provides additional intuition

**Complex Numbers** Before discussing the spectral density, we invite you to recall the main properties of complex numbers (or *skip to the next section*)

It can be helpful to remember that, in a formal sense, complex numbers are just points  $(x, y) \in \mathbb{R}^2$  endowed with a specific notion of multiplication

When  $(x, y)$  is regarded as a complex number,  $x$  is called the *real part* and  $y$  is called the *imaginary part*

The *modulus* or *absolute value* of a complex number  $z = (x, y)$  is just its Euclidean norm in  $\mathbb{R}^2$ , but is usually written as  $|z|$  instead of  $\|z\|$

The product of two complex numbers  $(x, y)$  and  $(u, v)$  is defined to be  $(xu - vy, xv + yu)$ , 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 can be thought of as 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  $(xu - vy, xv + yu)$ , which is the same as the product of  $(x, y)$  and  $(u, v)$  from our previous definition

Complex numbers are also sometimes expressed in their polar form  $re^{i\omega}$ , which should be interpreted as

$$re^{i\omega} := r(\cos(\omega) + i \sin(\omega))$$

**Spectral Densities** Let  $\{X_t\}$  be a covariance stationary process with autocovariance function  $\gamma$  satisfying  $\sum_k \gamma(k)^2 < \infty$

The *spectral density*  $f$  of  $\{X_t\}$  is defined as the [discrete time Fourier transform](#) of its autocovariance function  $\gamma$

$$f(\omega) := \sum_{k \in \mathbb{Z}} \gamma(k) e^{-i\omega k}, \quad \omega \in \mathbb{R}$$

(Some authors normalize the expression on the right by constants such as  $1/\pi$  — the chosen convention makes little difference provided you are consistent)

Using the fact that  $\gamma$  is *even*, in the sense that  $\gamma(t) = \gamma(-t)$  for all  $t$ , you should be able to show that

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k) \tag{3.64}$$

It is not difficult to confirm that  $f$  is

- real-valued
- even ( $f(\omega) = f(-\omega)$ ), and
- $2\pi$ -periodic, in the sense that  $f(2\pi + \omega) = f(\omega)$  for all  $\omega$

It follows that the values of  $f$  on  $[0, \pi]$  determine the values of  $f$  on all of  $\mathbb{R}$  — the proof is an exercise

For this reason it is standard to plot the spectral density only on the interval  $[0, \pi]$

**Example 1: White Noise** Consider a white noise process  $\{\epsilon_t\}$  with standard deviation  $\sigma$

It is simple to check that in this case we have  $f(\omega) = \sigma^2$ . In particular,  $f$  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

$$f(\omega) = \sigma^2(1 + 2\theta \cos(\omega) + \theta^2) \quad (3.65)$$

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

$$f(\omega) = \frac{\sigma^2}{1 - 2\phi \cos(\omega) + \phi^2} \quad (3.66)$$

More generally, it can be shown that the spectral density of the ARMA process (3.60) is

$$f(\omega) = \left| \frac{\theta(e^{i\omega})}{\phi(e^{i\omega})} \right|^2 \sigma^2 \quad (3.67)$$

where

- $\sigma$  is the standard deviation of the white noise process  $\{\epsilon_t\}$
- the polynomials  $\phi(\cdot)$  and  $\theta(\cdot)$  are as defined in (3.62)

The derivation of (3.67) 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 (3.65) and (3.66) are indeed special cases of (3.67)

**Interpreting the Spectral Density** Plotting (3.66) reveals the shape of the spectral density for the AR(1) model when  $\phi$  takes the values 0.8 and -0.8 respectively

These spectral densities correspond to the autocovariance functions for the AR(1) process *shown above*

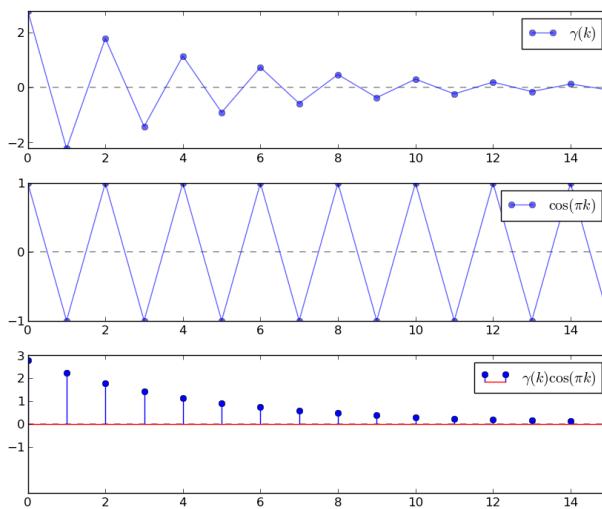
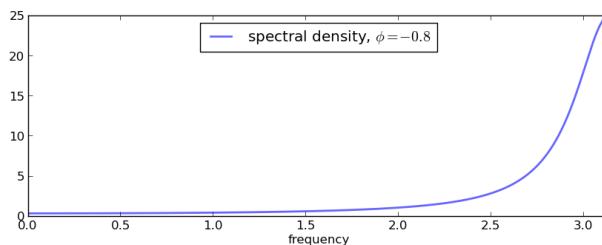
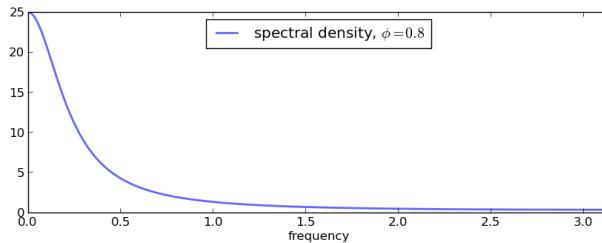
Informally, we think of the spectral density as being large at those  $\omega \in [0, \pi]$  such that the autocovariance function exhibits significant cycles at this “frequency”

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

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k) = \gamma(0) + 2 \sum_{k \geq 1} (-0.8)^k \cos(\omega k) \quad (3.68)$$

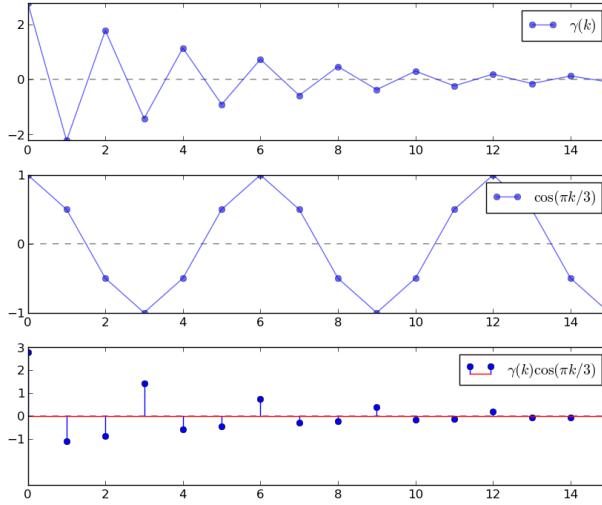
When we evaluate this at  $\omega = \pi$ , we get a large number because  $\cos(\pi k)$  is large and positive when  $(-0.8)^k$  is positive, and large in absolute value and negative when  $(-0.8)^k$  is negative



Hence the product is always large and positive, and hence the sum of the products on the right-hand side of (3.68) 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  $\omega$  where the autocovariance function exhibits 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

$$\gamma(k) = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(\omega) e^{i\omega k} d\omega \quad (3.69)$$

This is convenient in situations where the spectral density is easier to calculate and manipulate than the autocovariance function

(For example, the expression (3.67) 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

$$f = \sum_k \alpha_k h_k \quad \text{where} \quad \alpha_k := \langle f, h_k \rangle \quad (3.70)$$

where  $\langle \cdot, \cdot \rangle$  denotes the inner product in  $H$

Thus,  $f$  can be represented to any degree of precision by linearly combining basis vectors

The scalar sequence  $\alpha = \{\alpha_k\}$  is called the *Fourier coefficients* of  $f$ , and satisfies  $\sum_k |\alpha_k|^2 < \infty$

In other words,  $\alpha$  is in  $\ell_2$ , the set of square summable sequences

Consider an operator  $T$  that maps  $\alpha \in \ell_2$  into its expansion  $\sum_k \alpha_k h_k \in H$

The Fourier coefficients of  $T\alpha$  are just  $\alpha = \{\alpha_k\}$ , as you can verify by confirming that  $\langle T\alpha, h_k \rangle = \alpha_k$

Using elementary results from Hilbert space theory, it can be shown that

- $T$  is one-to-one — if  $\alpha$  and  $\beta$  are distinct in  $\ell_2$ , then so are their expansions in  $H$
- $T$  is onto — if  $f \in H$  then its preimage in  $\ell_2$  is the sequence  $\alpha$  given by  $\alpha_k = \langle f, h_k \rangle$
- $T$  is a linear isometry — in particular  $\langle \alpha, \beta \rangle = \langle T\alpha, T\beta \rangle$

Summarizing these results, we say that any separable Hilbert space is isometrically isomorphic to  $\ell_2$

In essence, this says that each separable Hilbert space we consider is just a different way of looking at the fundamental space  $\ell_2$

With this in mind, let's specialize to a setting where

- $\gamma \in \ell_2$  is the autocovariance function of a covariance stationary process, and  $f$  is the spectral density
- $H = L_2$ , where  $L_2$  is the set of square summable functions on the interval  $[-\pi, \pi]$ , with inner product  $\langle g, h \rangle = \int_{-\pi}^{\pi} g(\omega)h(\omega)d\omega$
- $\{h_k\} =$  the orthonormal basis for  $L_2$  given by the set of trigonometric functions

$$h_k(\omega) = \frac{e^{i\omega k}}{\sqrt{2\pi}}, \quad k \in \mathbb{Z}, \quad \omega \in [-\pi, \pi]$$

Using the definition of  $T$  from above and the fact that  $f$  is even, we now have

$$T\gamma = \sum_{k \in \mathbb{Z}} \gamma(k) \frac{e^{i\omega k}}{\sqrt{2\pi}} = \frac{1}{\sqrt{2\pi}} f(\omega) \quad (3.71)$$

In other words, apart from a scalar multiple, the spectral density is just a transformation of  $\gamma \in \ell_2$  under a certain linear isometry — a different way to view  $\gamma$

In particular, it is an expansion of the autocovariance function with respect to the trigonometric basis functions in  $L_2$

As discussed above, the Fourier coefficients of  $T\gamma$  are given by the sequence  $\gamma$ , and, in particular,  $\gamma(k) = \langle T\gamma, h_k \rangle$

Transforming this inner product into its integral expression and using (3.71) gives (3.69), 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.

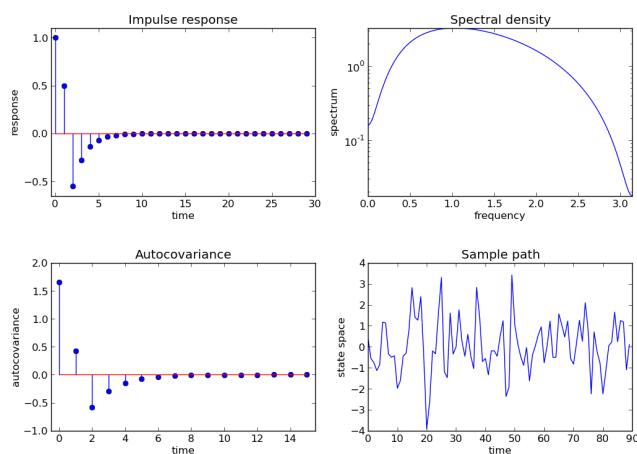
The module provides functions for mapping ARMA( $p, q$ ) models into their

1. impulse response function
2. simulated time series
3. autocovariance function
4. spectral density

In addition 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** For interest's sake, `arma.py` is printed below

```

"""
Filename: arma.py
Authors: Doc-Jin Jang, Jerry Choi, Thomas Sargent, John Stachurski

Provides functions for working with and visualizing scalar ARMA processes.

"""

import numpy as np
from numpy import conj, pi
import matplotlib.pyplot as plt
from scipy.signal import dimpulse, freqz, dlsim

# == Ignore unnecessary warnings concerning casting complex variables back to
# floats == #
import warnings
warnings.filterwarnings('ignore')

class ARMA(object):
    """
    This class represents scalar ARMA( $p, q$ ) processes.

    If  $\phi$  and  $\theta$  are scalars, then the model is
    understood to be

    ... math::

        X_t = \phi X_{t-1} + \epsilon_t + \theta \epsilon_{t-1}

    where :math:`\epsilon_t` is a white noise process with standard
    deviation :math:`\sigma`. If  $\phi$  and  $\theta$  are arrays or sequences,
    then the interpretation is the ARMA( $p, q$ ) model

    ... math::

        X_t = \phi_1 X_{t-1} + \dots + \phi_p X_{t-p} +
        \epsilon_t + \theta_1 \epsilon_{t-1} + \dots + \theta_q \epsilon_{t-q}

    where

    * :math:`\phi = (\phi_1, \phi_2, \dots, \phi_p)`
    * :math:`\theta = (\theta_1, \theta_2, \dots, \theta_q)`
    * :math:`\sigma` is a scalar, the standard deviation of the
      white noise

    Parameters
    -----
    phi : scalar or iterable or array_like(float)
        Autocorrelation values for the autocorrelated variable.
        See above for explanation.

```

```

theta : scalar or iterable or array_like(float)
    Autocorrelation values for the white noise of the model.
    See above for explanation
sigma : scalar(float)
    The standard deviation of the white noise

Attributes
-----
phi, theta, sigma : see Parameters
ar_poly : array_like(float)
    The polynomial form that is needed by scipy.signal to do the
    processing we desire. Corresponds with the phi values
ma_poly : array_like(float)
    The polynomial form that is needed by scipy.signal to do the
    processing we desire. Corresponds with the theta values

"""

def __init__(self, phi, theta=0, sigma=1):
    self._phi, self._theta = phi, theta
    self._sigma = sigma
    self.set_params()

def __repr__(self):
    m = "ARMA(phi=%s, theta=%s, sigma=%s)"
    return m % (self.phi, self.theta, self.sigma)

def __str__(self):
    m = "An ARMA({p}, {q}) process"
    p = np.asarray(self.phi).size
    q = np.asarray(self.theta).size
    return m.format(p=p, q=q)

# Special latex print method for working in notebook
def _repr_latex_(self):
    m = r"\$X_t = "
    phi = np.atleast_1d(self.phi)
    theta = np.atleast_1d(self.theta)
    rhs = ""
    for (tm, phi_p) in enumerate(phi):
        # don't include terms if they are equal to zero
        if abs(phi_p) > 1e-12:
            rhs += r"\%t^g X_{t-%i}" % (phi_p, tm+1)

    if rhs[0] == "+":
        rhs = rhs[1:] # remove initial '+' if phi_1 was positive

    rhs += r" + \epsilon_t"

    for (tm, th_q) in enumerate(theta):
        # don't include terms if they are equal to zero
        if abs(th_q) > 1e-12:
            rhs += r"\%t^g \epsilon_{t-%i}" % (th_q, tm+1)

```

```

    return m + rhs + " $""

@property
def phi(self):
    return self._phi

@phi.setter
def phi(self, new_value):
    self._phi = new_value
    self.set_params()

@property
def theta(self):
    return self._theta

@theta.setter
def theta(self, new_value):
    self._theta = new_value
    self.set_params()

def set_params(self):
    """
    Internally, scipy.signal works with systems of the form

    .. math::

        ar\_poly(L) X_t = ma\_poly(L) \epsilon_t

    where L is the lag operator. To match this, we set

    .. math::

        ar\_poly = (1, -\phi_1, -\phi_2, \dots, -\phi_p)

        ma\_poly = (1, \theta_1, \theta_2, \dots, \theta_q)

    In addition, ar_poly must be at least as long as ma_poly.
    This can be achieved by padding it out with zeros when required.

    """
    # === set up ma_poly === #
    ma_poly = np.asarray(self._theta)
    self.ma_poly = np.insert(ma_poly, 0, 1) # The array (1, theta)

    # === set up ar_poly === #
    if np.isscalar(self._phi):
        ar_poly = np.array(-self._phi)
    else:
        ar_poly = -np.asarray(self._phi)
    self.ar_poly = np.insert(ar_poly, 0, 1) # The array (1, -phi)

    # === pad ar_poly with zeros if required === #
    if len(self.ar_poly) < len(self.ma_poly):

```

```

        temp = np.zeros(len(self.ma_poly) - len(self.ar_poly))
        self.ar_poly = np.hstack((self.ar_poly, temp))

    def impulse_response(self, impulse_length=30):
        """
        Get the impulse response corresponding to our model.

        Returns
        ----
        psi : array_like(float)
            psi[j] is the response at lag j of the impulse response.
            We take psi[0] as unity.

        """
        sys = self.ma_poly, self.ar_poly, 1
        times, psi = dimpulse(sys, n=impulse_length)
        psi = psi[0].flatten() # Simplify return value into flat array

        return psi

    def spectral_density(self, two_pi=True, res=1200):
        """
        Compute the spectral density function. The spectral density is
        the discrete time Fourier transform of the autocovariance
        function. In particular,

        .. math::

            f(w) = \sum_k \gamma(k) \exp(-ikw)

        where gamma is the autocovariance function and the sum is over
        the set of all integers.

        Parameters
        -----
        two_pi : Boolean, optional
            Compute the spectral density function over [0, pi] if
            two_pi is False and [0, 2 pi] otherwise. Default value is
            True
        res : scalar or array_like(int), optional(default=1200)
            If res is a scalar then the spectral density is computed at
            `res` frequencies evenly spaced around the unit circle, but
            if res is an array then the function computes the response
            at the frequencies given by the array

        Returns
        -----
        w : array_like(float)
            The normalized frequencies at which h was computed, in
            radians/sample
        spect : array_like(float)
            The frequency response

```

```

    """
    w, h = freqz(self.ma_poly, self.ar_poly, worN=res, whole=two_pi)
    spect = h * conj(h) * self.sigma**2

    return w, spect

def autocovariance(self, num_autocov=16):
    """
    Compute the autocovariance function from the ARMA parameters
    over the integers range(num_autocov) using the spectral density
    and the inverse Fourier transform.

    Parameters
    -----
    num_autocov : scalar(int), optional(default=16)
        The number of autocovariances to calculate

    """
    spect = self.spectral_density()[1]
    acov = np.fft.ifft(spect).real

    # num_autocov should be <= len(acov) / 2
    return acov[:num_autocov]

def simulation(self, ts_length=90):
    """
    Compute a simulated sample path assuming Gaussian shocks.

    Parameters
    -----
    ts_length : scalar(int), optional(default=90)
        Number of periods to simulate for

    Returns
    -----
    vals : array_like(float)
        A simulation of the model that corresponds to this class

    """
    sys = self.ma_poly, self.ar_poly, 1
    u = np.random.randn(ts_length, 1) * self.sigma
    vals = dlsim(sys, u)[1]

    return vals.flatten()

def plot_impulse_response(self, ax=None, show=True):
    if show:
        fig, ax = plt.subplots()
        ax.set_title('Impulse response')
        yi = self.impulse_response()
        ax.stem(list(range(len(yi))), yi)
        ax.set_xlim(xmin=-0.5)
        ax.set_ylim(min(yi)-0.1, max(yi)+0.1)

```

```

        ax.set_xlabel('time')
        ax.set_ylabel('response')
        if show:
            plt.show()

    def plot_spectral_density(self, ax=None, show=True):
        if show:
            fig, ax = plt.subplots()
        ax.set_title('Spectral density')
        w, spect = self.spectral_density(two_pi=False)
        ax.semilogy(w, spect)
        ax.set_xlim(0, pi)
        ax.set_ylimits(0, np.max(spect))
        ax.set_xlabel('frequency')
        ax.set_ylabel('spectrum')
        if show:
            plt.show()

    def plot_autocovariance(self, ax=None, show=True):
        if show:
            fig, ax = plt.subplots()
        ax.set_title('Autocovariance')
        acov = self.autocovariance()
        ax.stem(list(range(len(acov))), acov)
        ax.set_xlim(-0.5, len(acov) - 0.5)
        ax.set_xlabel('time')
        ax.set_ylabel('autocovariance')
        if show:
            plt.show()

    def plot_simulation(self, ax=None, show=True):
        if show:
            fig, ax = plt.subplots()
        ax.set_title('Sample path')
        x_out = self.simulation()
        ax.plot(x_out)
        ax.set_xlabel('time')
        ax.set_ylabel('state space')
        if show:
            plt.show()

    def quad_plot(self):
        """
        Plots the impulse response, spectral density, autocovariance,
        and one realization of the process.

        """
        num_rows, num_cols = 2, 2
        fig, axes = plt.subplots(num_rows, num_cols, figsize=(12, 8))
        plt.subplots_adjust(hspace=0.4)
        plot_functions = [self.plot_impulse_response,
                          self.plot_spectral_density,
                          self.plot_autocovariance,

```

```

        self.plot_simulation]
    for plot_func, ax in zip(plot_functions, axes.flatten()):
        plot_func(ax, show=False)
    plt.show()

```

Here's an example of usage

```

In [1]: import quantecon as qe
In [2]: phi = 0.5
In [3]: theta = 0, -0.8
In [4]: lp = qe.ARMA(phi, theta)
In [5]: lp.quad_plot()

```

**Explanation** The call

```
lp = ARMA(phi, theta, sigma)
```

creates an instance `lp` that represents the ARMA( $p, q$ ) model

$$X_t = \phi_1 X_{t-1} + \dots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \dots + \theta_q \epsilon_{t-q}$$

If `phi` and `theta` are arrays or sequences, then the interpretation will be

- `phi` holds the vector of parameters  $(\phi_1, \phi_2, \dots, \phi_p)$
- `theta` holds the vector of parameters  $(\theta_1, \theta_2, \dots, \theta_q)$

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 in to its functions in a manner consistent with the alternative ARMA notation (3.63)

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 (3.62) — that is,

- `ma_poly` is the vector  $(1, \theta_1, \theta_2, \dots, \theta_q)$
- `ar_poly` is the vector  $(1, -\phi_1, -\phi_2, \dots, -\phi_p)$

To this end, we also maintain the arrays `ma_poly` and `ar_poly` as instance data, with their values computed automatically from the values of `phi` and `theta` supplied by the user

If the user decides to change the value of either `phi` or `theta` ex-post by assignments such as

```
lp.phi = (0.5, 0.2)
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  $A_0, A_1, \dots, A_{n-1}$  and returns the sequence  $a_0, a_1, \dots, a_{n-1}$  defined by

$$a_k = \frac{1}{n} \sum_{t=0}^{n-1} A_t e^{ik2\pi t/n}$$

Thus, if we set  $A_t = f(\omega_t)$ , where  $f$  is the spectral density and  $\omega_t := 2\pi t/n$ , then

$$a_k = \frac{1}{n} \sum_{t=0}^{n-1} f(\omega_t) e^{i\omega_t k} = \frac{1}{2\pi} \frac{2\pi}{n} \sum_{t=0}^{n-1} f(\omega_t) e^{i\omega_t k}, \quad \omega_t := 2\pi t/n$$

For  $n$  sufficiently large, we then have

$$a_k \approx \frac{1}{2\pi} \int_0^{2\pi} f(\omega) e^{i\omega k} d\omega = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(\omega) e^{i\omega k} d\omega$$

(You can check the last equality)

In view of (3.69) 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  $\gamma$  can be written as

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k), \quad \omega \in \mathbb{R}$$

Now consider the problem of estimating the spectral density of a given time series, when  $\gamma$  is unknown

In particular, let  $X_0, \dots, X_{n-1}$  be  $n$  consecutive observations of a single time series that is assumed to be covariance stationary

The most common estimator of the spectral density of this process is the *periodogram* of  $X_0, \dots, X_{n-1}$ , which is defined as

$$I(\omega) := \frac{1}{n} \left| \sum_{t=0}^{n-1} X_t e^{it\omega} \right|^2, \quad \omega \in \mathbb{R} \tag{3.72}$$

(Recall that  $|z|$  denotes the modulus of complex number  $z$ )

Alternatively,  $I(\omega)$  can be expressed as

$$I(\omega) = \frac{1}{n} \left\{ \left[ \sum_{t=0}^{n-1} X_t \cos(\omega t) \right]^2 + \left[ \sum_{t=0}^{n-1} X_t \sin(\omega t) \right]^2 \right\}$$

It is straightforward to show that the function  $I$  is even and  $2\pi$ -periodic (i.e.,  $I(\omega) = I(-\omega)$  and  $I(\omega + 2\pi) = I(\omega)$  for all  $\omega \in \mathbb{R}$ )

From these two results, you will be able to verify that the values of  $I$  on  $[0, \pi]$  determine the values of  $I$  on all of  $\mathbb{R}$

The next section helps to explain the connection between the periodogram and the spectral density

**Interpretation** To interpret the periodogram, it is convenient to focus on its values at the *Fourier frequencies*

$$\omega_j := \frac{2\pi j}{n}, \quad j = 0, \dots, n-1$$

In what sense is  $I(\omega_j)$  an estimate of  $f(\omega_j)$ ?

The answer is straightforward, although it does involve some algebra

With a bit of effort one can show that, for any integer  $j > 0$ ,

$$\sum_{t=0}^{n-1} e^{it\omega_j} = \sum_{t=0}^{n-1} \exp \left\{ i2\pi j \frac{t}{n} \right\} = 0$$

Letting  $\bar{X}$  denote the sample mean  $n^{-1} \sum_{t=0}^{n-1} X_t$ , we then have

$$nI(\omega_j) = \left| \sum_{t=0}^{n-1} (X_t - \bar{X}) e^{it\omega_j} \right|^2 = \sum_{t=0}^{n-1} (X_t - \bar{X}) e^{it\omega_j} \sum_{r=0}^{n-1} (X_r - \bar{X}) e^{-ir\omega_j}$$

By carefully working through the sums, one can transform this to

$$nI(\omega_j) = \sum_{t=0}^{n-1} (X_t - \bar{X})^2 + 2 \sum_{k=1}^{n-1} \sum_{t=k}^{n-1} (X_t - \bar{X})(X_{t-k} - \bar{X}) \cos(\omega_j k)$$

Now let

$$\hat{\gamma}(k) := \frac{1}{n} \sum_{t=k}^{n-1} (X_t - \bar{X})(X_{t-k} - \bar{X}), \quad k = 0, 1, \dots, n-1$$

This is the sample autocovariance function, the natural “plug-in estimator” of the *autocovariance function*  $\gamma$

(“Plug-in estimator” is an informal term for an estimator found by replacing expectations with sample means)

With this notation, we can now write

$$I(\omega_j) = \hat{\gamma}(0) + 2 \sum_{k=1}^{n-1} \hat{\gamma}(k) \cos(\omega_j k)$$

Recalling our expression for  $f$  given *above*, we see that  $I(\omega_j)$  is just a sample analog of  $f(\omega_j)$

**Calculation** Let’s now consider how to compute the periodogram as defined in (3.72)

There are already functions available that will do this for us — an example is `statsmodels.tsa.stattools.periodogram` in the `statsmodels` package

However, it is very simple to replicate their results, and this will give us a platform to make useful extensions

The most common way to calculate the periodogram is via the discrete Fourier transform, which in turn is implemented through the `fast Fourier transform` algorithm

In general, given a sequence  $a_0, \dots, a_{n-1}$ , the discrete Fourier transform computes the sequence

$$A_j := \sum_{t=0}^{n-1} a_t \exp \left\{ i2\pi \frac{tj}{n} \right\}, \quad j = 0, \dots, n-1$$

With `numpy.fft.fft` imported as `fft` and  $a_0, \dots, a_{n-1}$  stored in NumPy array `a`, the function call `fft(a)` returns the values  $A_0, \dots, A_{n-1}$  as a NumPy array

It follows that, when the data  $X_0, \dots, X_{n-1}$  is stored in array `X`, the values  $I(\omega_j)$  at the Fourier frequencies, which are given by

$$\frac{1}{n} \left| \sum_{t=0}^{n-1} X_t \exp \left\{ i2\pi \frac{tj}{n} \right\} \right|^2, \quad j = 0, \dots, n-1$$

can be computed by `np.abs(fft(X))**2 / len(X)`

Note: The NumPy function `abs` acts elementwise, and correctly handles complex numbers (by computing their modulus, which is exactly what we need)

Here's a function that puts all this together

```
import numpy as np
from numpy.fft import fft

def periodogram(x):
    "Argument x is a NumPy array containing the time series data"
    n = len(x)
    I_w = np.abs(fft(x))**2 / n
    w = 2 * np.pi * np.arange(n) / n      # Fourier frequencies
    w, I_w = w[:int(n/2)], I_w[:int(n/2)] # Truncate to interval [0, pi]
    return w, I_w
```

Let's generate some data for this function using the `ARMA` class from `QuantEcon`

(See the *lecture on linear processes* for details on this class)

Here's a code snippet that, once the preceding code has been run, generates data from the process

$$X_t = 0.5X_{t-1} + \epsilon_t - 0.8\epsilon_{t-2} \quad (3.73)$$

where  $\{\epsilon_t\}$  is white noise with unit variance, and compares the periodogram to the actual spectral density

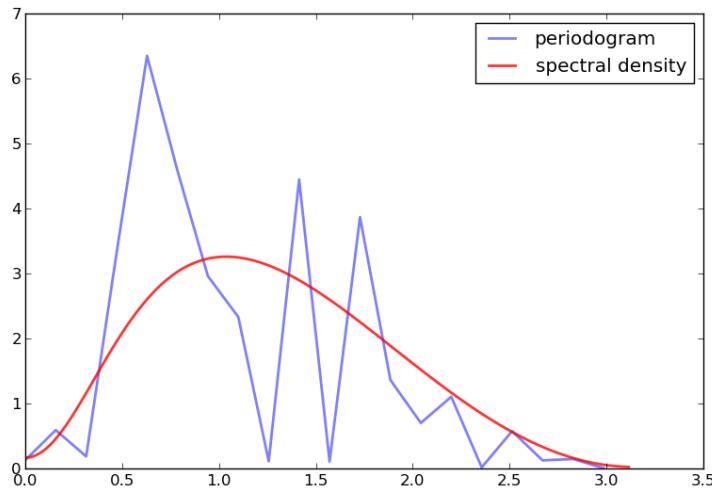
```
import matplotlib.pyplot as plt
from quantecon import ARMA

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')
```

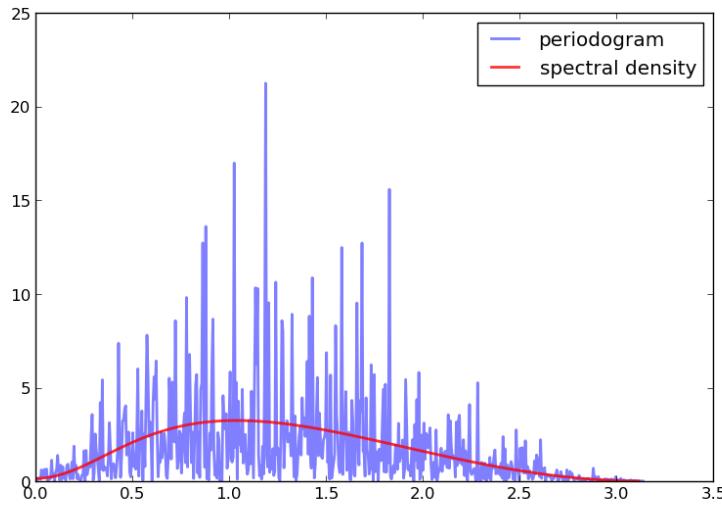
```
x_sd, y_sd = lp.spectral_density(two_pi=False, resolution=120)
ax.plot(x_sd, y_sd, 'r-', lw=2, alpha=0.8, label='spectral density')
ax.legend()
plt.show()
```

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  $\omega$  at which  $I(\omega)$  is estimated increases in line with the amount of data

In other words, although we have more data, we are also using it to estimate more values

A second issue is that densities of all types are fundamentally hard to estimate without parametric assumptions

Typically, nonparametric estimation of densities requires some degree of smoothing

The standard way that smoothing is applied to periodograms is by taking local averages

In other words, the value  $I(\omega_j)$  is replaced with a weighted average of the adjacent values

$$I(\omega_{j-p}), I(\omega_{j-p+1}), \dots, I(\omega_j), \dots, I(\omega_{j+p})$$

This weighted average can be written as

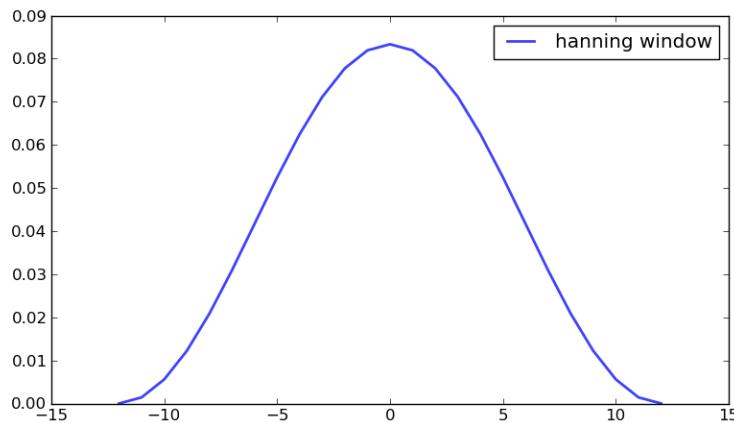
$$I_S(\omega_j) := \sum_{\ell=-p}^p w(\ell) I(\omega_{j+\ell}) \quad (3.74)$$

where the weights  $w(-p), \dots, w(p)$  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 (3.74)



**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 via `QuantEcon.py`

The file `estspec.py` are printed below

```
"""
Filename: estspec.py

Authors: Thomas Sargent, John Stachurski

Functions for working with periodograms of scalar data.

"""

from __future__ import division, print_function
import numpy as np
from numpy.fft import fft
from pandas import ols, Series


def smooth(x, window_len=7, window='hanning'):
    """
    Smooth the data in x using convolution with a window of requested
    size and type.

    Parameters
    -----
    x : array_like(float)
        A flat NumPy array containing the data to smooth
    window_len : scalar(int), optional
        An odd integer giving the length of the window. Defaults to 7.
    window : string
        A string giving the window type. Possible values are 'flat',
        'hanning', 'hamming', 'bartlett' or 'blackman'

    Returns
    -----
    array_like(float)
        The smoothed values

    Notes
    -----
    Application of the smoothing window at the top and bottom of x is
    done by reflecting x around these points to extend it sufficiently
    in each direction.

    """
    if len(x) < window_len:
        raise ValueError("Input vector length must be >= window length.")

    if window_len < 3:
        raise ValueError("Window length must be at least 3.")

    if not window_len % 2: # window_len is even
        window_len += 1
```

```

    print("Window length reset to {}".format(window_len))

windows = {'hanning': np.hanning,
           'hamming': np.hamming,
           'bartlett': np.bartlett,
           'blackman': np.blackman,
           'flat': np.ones # moving average
          }

# === Reflect x around x[0] and x[-1] prior to convolution === #
k = int(window_len / 2)
xb = x[:k] # First k elements
xt = x[-k:] # Last k elements
s = np.concatenate((xb[::-1], x, xt[::-1]))

# === Select window values === #
if window in windows.keys():
    w = windows[window](window_len)
else:
    msg = "Unrecognized window type '{}'".format(window)
    print(msg + " Defaulting to hanning")
    w = windows['hanning'](window_len)

return np.convolve(w / w.sum(), s, mode='valid')

def periodogram(x, window=None, window_len=7):
    """
    Computes the periodogram

    .. math::

        I(w) = (1 / n) | \sum_{t=0}^{n-1} x_t e^{-itw} |^2

    at the Fourier frequencies  $w_j := 2 \pi j / n$ ,  $j = 0, \dots, n - 1$ ,
    using the fast Fourier transform. Only the frequencies  $w_j$  in  $[0,$ 
 $\pi]$  and corresponding values  $I(w_j)$  are returned. If a window type
    is given then smoothing is performed.

    Parameters
    -----
    x : array_like(float)
        A flat NumPy array containing the data to smooth
    window_len : scalar(int), optional(default=7)
        An odd integer giving the length of the window. Defaults to 7.
    window : string
        A string giving the window type. Possible values are 'flat',
        'hanning', 'hamming', 'bartlett' or 'blackman'

    Returns
    -----
    w : array_like(float)
        Fourier frequencies at which periodogram is evaluated
    """

```

```

I_w : array_like(float)
    Values of periodogram at the Fourier frequencies

"""
n = len(x)
I_w = np.abs(fft(x))**2 / n
w = 2 * np.pi * np.arange(n) / n  # Fourier frequencies
w, I_w = w[:int(n/2)+1], I_w[:int(n/2)+1]  # Take only values on [0, pi]
if window:
    I_w = smooth(I_w, window_len=window_len, window=window)
return w, I_w

def ar_periodogram(x, window='hanning', window_len=7):
    """
    Compute periodogram from data x, using prewhitening, smoothing and
    recoloring. The data is fitted to an AR(1) model for prewhitening,
    and the residuals are used to compute a first-pass periodogram with
    smoothing. The fitted coefficients are then used for recoloring.

    Parameters
    -----
    x : array_like(float)
        A flat NumPy array containing the data to smooth
    window_len : scalar(int), optional
        An odd integer giving the length of the window. Defaults to 7.
    window : string
        A string giving the window type. Possible values are 'flat',
        'hanning', 'hamming', 'bartlett' or 'blackman'

    Returns
    -----
    w : array_like(float)
        Fourier frequencies at which periodogram is evaluated
    I_w : array_like(float)
        Values of periodogram at the Fourier frequencies

    """
# === run regression === #
x_current, x_lagged = x[1:], x[:-1]  # x_t and x_{t-1}
x_current, x_lagged = Series(x_current), Series(x_lagged)  # pandas series
results = ols(y=x_current, x=x_lagged, intercept=True, nw_lags=1)
e_hat = results.resid.values
phi = results.beta['x']

# === compute periodogram on residuals === #
w, I_w = periodogram(e_hat, window=window, window_len=window_len)

# === recolor and return === #
I_w = I_w / np.abs(1 - phi * np.exp(1j * w))**2

return w, I_w

```

The 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 to either explore or simply use this code according to their interests

The next three figures each show smoothed and unsmoothed periodograms, as well as the true spectral density

(The model is the same as before — see equation (3.73) — 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 *above* we showed three functions from the file `estspec.py`

The third function in the file (`ar_periodogram()`) adds a pre-processing step to periodogram smoothing

First we describe the basic idea, and after that we give the code

The essential idea is to

1. Transform the data in order to make estimation of the spectral density more efficient
2. Compute the periodogram associated with the transformed data
3. Reverse the effect of the transformation on the periodogram, so that it now estimates the spectral density of the original process

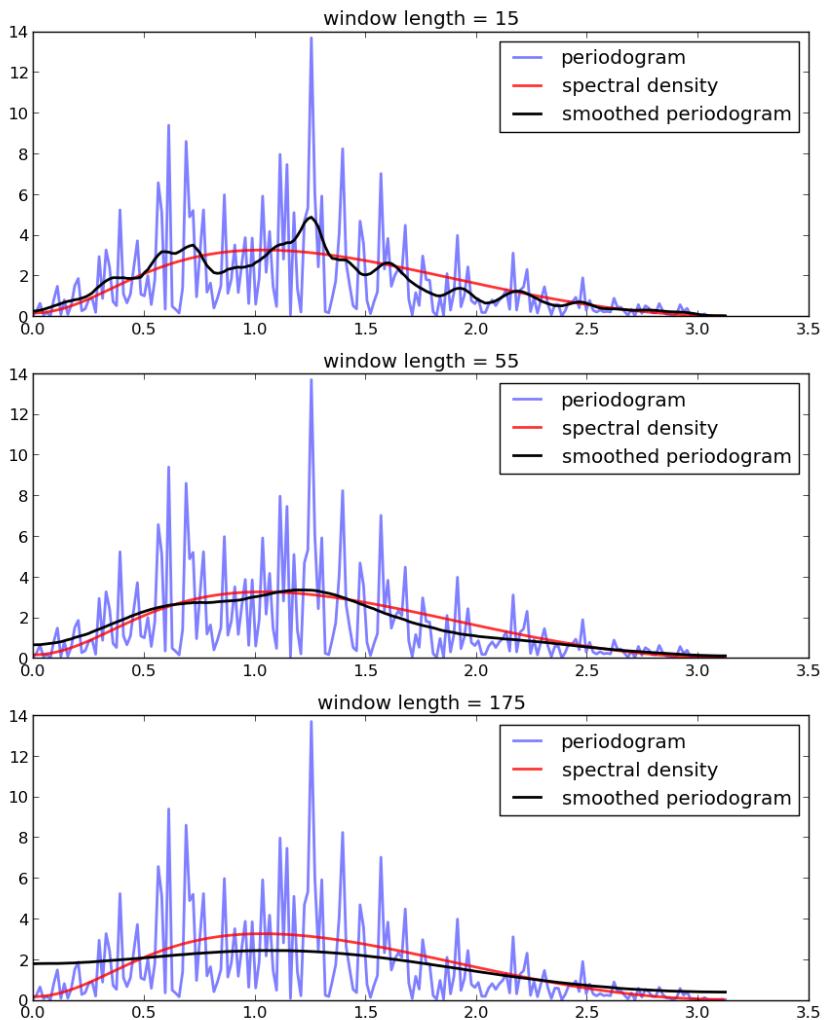
Step 1 is called *pre-filtering* or *pre-whitening*, while step 3 is called *recoloring*

The first step is called pre-whitening because the transformation is usually designed to turn the data into something closer to white noise

Why would this be desirable in terms of spectral density estimation?

The reason is that we are smoothing our estimated periodogram based on estimated values at nearby points — recall (3.74)

The underlying assumption that makes this a good idea is that the true spectral density is relatively regular — the value of  $I(\omega)$  is close to that of  $I(\omega')$  when  $\omega$  is close to  $\omega'$



This will not be true in all cases, but it is certainly true for white noise

For white noise,  $I$  is as regular as possible — *it is a constant function*

In this case, values of  $I(\omega')$  at points  $\omega'$  near to  $\omega$  provided the maximum possible amount of information about the value  $I(\omega)$

Another way to put this is that if  $I$  is relatively constant, then we can use a large amount of smoothing without introducing too much bias

**The AR(1) Setting** Let's examine this idea more carefully in a particular setting — where the data is assumed to be AR(1)

(More general ARMA settings can be handled using similar techniques to those described below)

Suppose in particular that  $\{X_t\}$  is covariance stationary and AR(1), with

$$X_{t+1} = \mu + \phi X_t + \epsilon_{t+1} \quad (3.75)$$

where  $\mu$  and  $\phi \in (-1, 1)$  are unknown parameters and  $\{\epsilon_t\}$  is white noise

It follows that if we regress  $X_{t+1}$  on  $X_t$  and an intercept, the residuals will approximate white noise

Let

- $g$  be the spectral density of  $\{\epsilon_t\}$  — a constant function, as discussed above
- $I_0$  be the periodogram estimated from the residuals — an estimate of  $g$
- $f$  be the spectral density of  $\{X_t\}$  — the object we are trying to estimate

In view of *an earlier result* we obtained while discussing ARMA processes,  $f$  and  $g$  are related by

$$f(\omega) = \left| \frac{1}{1 - \phi e^{i\omega}} \right|^2 g(\omega) \quad (3.76)$$

This suggests that the recoloring step, which constructs an estimate  $I$  of  $f$  from  $I_0$ , should set

$$I(\omega) = \left| \frac{1}{1 - \hat{\phi} e^{i\omega}} \right|^2 I_0(\omega)$$

where  $\hat{\phi}$  is the OLS estimate of  $\phi$

The code for `ar_periodogram()` — the third function in `estspec.py` — does exactly this. (See the code [here](#))

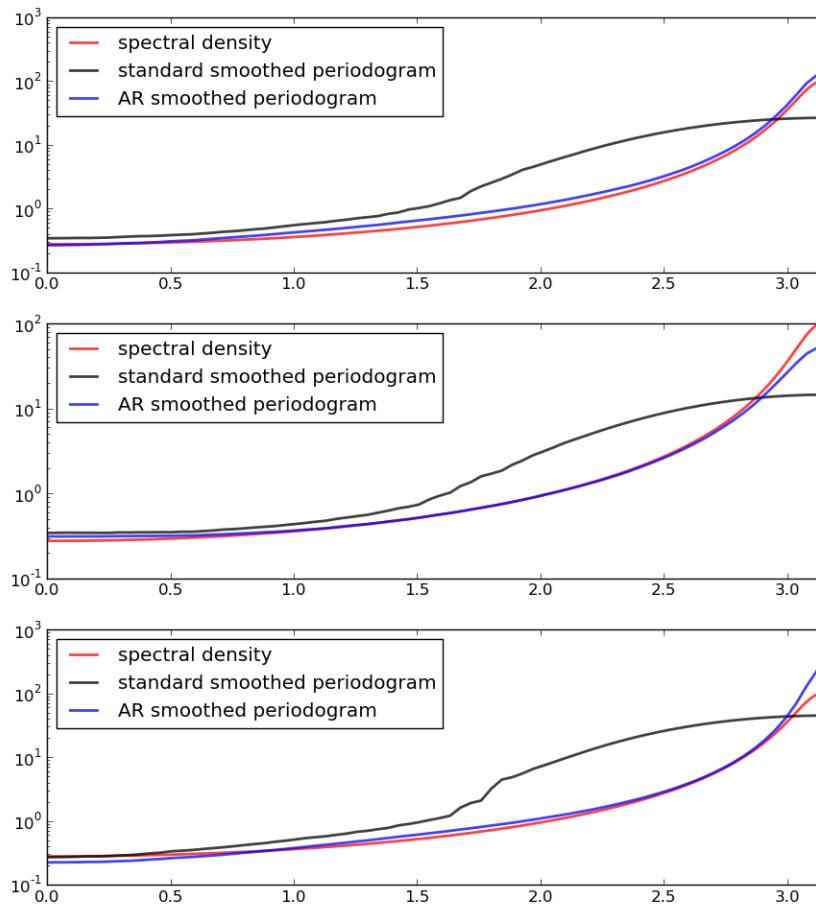
The next figure shows realizations of the two kinds of smoothed periodograms

1. “standard smoothed periodogram”, the ordinary smoothed periodogram, and
2. “AR smoothed periodogram”, the pre-whitened and recolored one generated by `ar_periodogram()`

The periodograms are calculated from time series drawn from (3.75) 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 (3.73) and there are 400 observations

For the smoothed periodogram, the window type is “hamming”

**Exercise 2** Replicate *this figure* (modulo randomness)

The model is as in equation (3.75), 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

[Solution notebook](#)

## 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

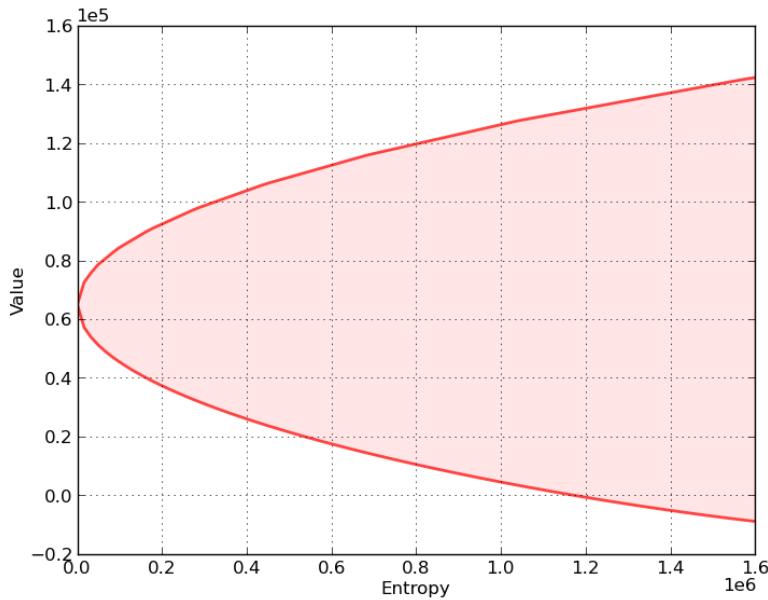
With this in mind, consider the following graph, which relates to a particular decision problem to be explained below

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

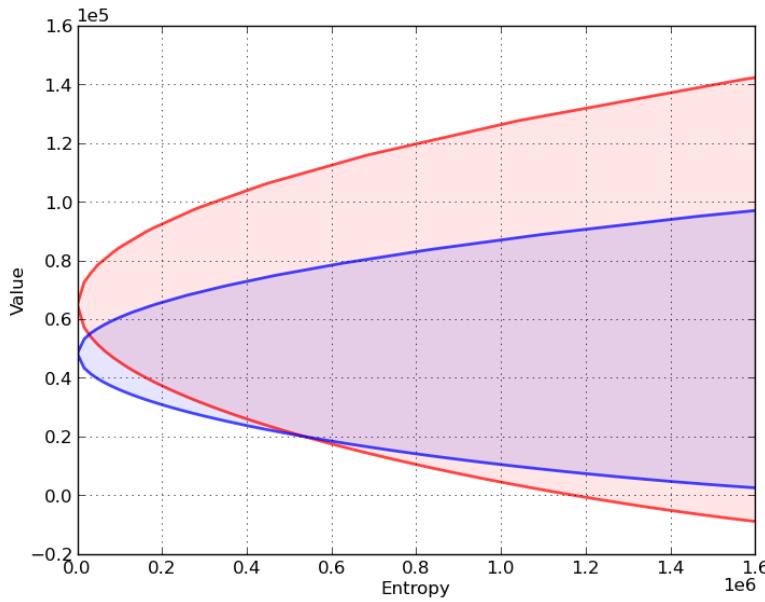
This conveys the sense in which the decision rule  $F_b$  is *more robust* than the decision rule  $F_r$

- *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

$$\sum_{t=0}^{\infty} \beta^t \{x_t' Rx_t + u_t' Qu_t\} \quad (3.77)$$

subject to the linear law of motion

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \quad t = 0, 1, 2, \dots \quad (3.78)$$

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

$$J(x) = \min_u \max_w \{x' Rx + u' Qu + \beta [J(Ax + Bu + Cw) - \theta w' w]\} \quad (3.79)$$

Notice that we've added the penalty term  $-\theta w' w$

Since  $w'w = \|w\|^2$ , this term becomes influential when  $w$  moves away from the origin

The penalty parameter  $\theta$  controls how much we penalize the maximizing agent for “harming” the minimizing agent

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

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

**Analyzing the Bellman equation** So what does  $J$  in (3.79) 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

$$\begin{aligned} \max_w \{(Ax + Bu + Cw)'P(Ax + Bu + Cw) - \theta w'w\} \\ = (Ax + Bu)'D(P)(Ax + Bu) \end{aligned} \quad (3.80)$$

where

$$D(P) := P + PC(\theta I - C'PC)^{-1}C'P \quad (3.81)$$

and  $I$  is a  $j \times j$  identity matrix. Substituting this expression for the maximum into (3.79) yields

$$x'Px = \min_u \{x'Rx + u'Qu + \beta(Ax + Bu)'D(P)(Ax + Bu)\} \quad (3.82)$$

Using similar mathematics, the solution to this minimization problem is  $u = -Fx$  where  $F := (Q + \beta B'D(P)B)^{-1}\beta B'D(P)A$

Substituting this minimizer back into (3.82) and working through the algebra gives  $x'Px = x'B(D(P))x$  for all  $x$ , or, equivalently,

$$P = B(D(P))$$

where  $D$  is the operator defined in (3.81) and

$$B(P) := R - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA + \beta A'PA$$

The operator  $B$  is the standard (i.e., non-robust) LQ Bellman operator, and  $P = B(P)$  is the standard matrix Riccati equation coming from the Bellman equation — see *this discussion*

Under some regularity conditions (see [HS08]), the operator  $B \circ D$  has a unique positive definite fixed point, which we denote below by  $\hat{P}$

A robust policy, indexed by  $\theta$ , is  $u = -\hat{F}x$  where

$$\hat{F} := (Q + \beta B'D(\hat{P})B)^{-1}\beta B'D(\hat{P})A \quad (3.83)$$

We also define

$$\hat{K} := (\theta I - C' \hat{P} C)^{-1} C' \hat{P} (A - B \hat{F}) \quad (3.84)$$

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 (3.80) evaluated at the fixed rule  $u = -\hat{F}x$

Note that  $\hat{P}, \hat{F}, \hat{K}$  are all determined by the primitives and  $\theta$

Note also that if  $\theta$  is very large, then  $\mathcal{D}$  is approximately equal to the identity mapping

Hence, when  $\theta$  is large,  $\hat{P}$  and  $\hat{F}$  are approximately equal to their standard LQ values

Furthermore, when  $\theta$  is large,  $\hat{K}$  is approximately equal to zero

Conversely, smaller  $\theta$  is associated with greater fear of model misspecification, and greater concern for robustness

### 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  $\{0, 0, 0, \dots\}$

A natural way to say "sufficiently close to the zero sequence" is to restrict the summed inner product  $\sum_{t=1}^{\infty} w'_t w_t$  to be small

However, to obtain a time-invariant recursive formulation, it turns out to be convenient to restrict a discounted inner product

$$\sum_{t=1}^{\infty} \beta^t w'_t w_t \leq \eta \quad (3.85)$$

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 (3.77), this value can be written as

$$J_F(x_0, \mathbf{w}) := \sum_{t=0}^{\infty} \beta^t x'_t (R + F' Q F) x_t \quad (3.86)$$

where

$$x_{t+1} = (A - BF)x_t + Cw_{t+1} \quad (3.87)$$

and the initial condition  $x_0$  is as specified in the left side of (3.86)

Agent 2 chooses  $\mathbf{w}$  to maximize agent 1's loss  $J_F(x_0, \mathbf{w})$  subject to (3.85)

Using a Lagrangian formulation, we can express this problem as

$$\max_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \{ x'_t (R + F' QF) x_t - \beta \theta (w'_{t+1} w_{t+1} - \eta) \}$$

where  $\{x_t\}$  satisfied (3.87) and  $\theta$  is a Lagrange multiplier on constraint (3.85)

For the moment, let's take  $\theta$  as fixed, allowing us to drop the constant  $\beta \theta \eta$  term in the objective function, and hence write the problem as

$$\max_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \{ x'_t (R + F' QF) x_t - \beta \theta w'_{t+1} w_{t+1} \}$$

or, equivalently,

$$\min_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \{ -x'_t (R + F' QF) x_t + \beta \theta w'_{t+1} w_{t+1} \} \quad (3.88)$$

subject to (3.87)

What's striking about this optimization problem is that it is once again an LQ discounted dynamic programming problem, with  $\mathbf{w} = \{w_t\}$  as the sequence of controls

The expression for the optimal policy can be found by applying the usual LQ formula (*see here*)

We denote it by  $K(F, \theta)$ , with the interpretation  $w_{t+1} = K(F, \theta)x_t$

The remaining step for agent 2's problem is to set  $\theta$  to enforce the constraint (3.85), which can be done by choosing  $\theta = \theta_\eta$  such that

$$\beta \sum_{t=0}^{\infty} \beta^t x'_t K(F, \theta_\eta)' K(F, \theta_\eta) x_t = \eta \quad (3.89)$$

Here  $x_t$  is given by (3.87) — 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 (3.88) as  $R_\theta(x_0, F)$ .

Because "minimizers minimize" we have

$$R_\theta(x_0, F) \leq \sum_{t=0}^{\infty} \beta^t \{ -x'_t (R + F' QF) x_t \} + \beta \theta \sum_{t=0}^{\infty} \beta^t w'_{t+1} w_{t+1},$$

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

$$R_\theta(x_0, F) - \theta \text{ ent} \leq \sum_{t=0}^{\infty} \beta^t \{-x_t'(R + F'QF)x_t\} \quad (3.90)$$

where

$$\text{ent} := \beta \sum_{t=0}^{\infty} \beta^t w_{t+1}' w_{t+1}$$

The left side of inequality (3.90) 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 (3.90) is attained when

$$\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \theta)' K(F, \theta) x_t \quad (3.91)$$

To construct the *lower bound* on the set of values associated with all perturbations  $\mathbf{w}$  satisfying the entropy constraint (3.85) at a given entropy level, we proceed as follows:

- For a given  $\theta$ , solve the minimization problem (3.88)
- Compute the minimizer  $R_\theta(x_0, F)$  and the associated entropy using (3.91)
- 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  $\theta$  to trace out the lower bound

**Note:** This procedure sweeps out a set of separating hyperplanes indexed by different values for the Lagrange multiplier  $\theta$

**The Upper Bound** To construct an *upper bound* we use a very similar procedure

We simply replace the *minimization* problem (3.88) with the *maximization* problem

$$V_{\tilde{\theta}}(x_0, F) = \max_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \{-x_t'(R + F'QF)x_t - \beta \tilde{\theta} w_{t+1}' w_{t+1}\} \quad (3.92)$$

where now  $\tilde{\theta} > 0$  penalizes the choice of  $\mathbf{w}$  with larger entropy.

(Notice that  $\tilde{\theta} = -\theta$  in problem (3.88))

Because “maximizers maximize” we have

$$V_{\tilde{\theta}}(x_0, F) \geq \sum_{t=0}^{\infty} \beta^t \{-x_t'(R + F'QF)x_t\} - \beta \tilde{\theta} \sum_{t=0}^{\infty} \beta^t w_{t+1}' w_{t+1}$$

which in turn implies the inequality

$$V_{\tilde{\theta}}(x_0, F) + \tilde{\theta} \text{ ent} \geq \sum_{t=0}^{\infty} \beta^t \{-x_t'(R + F'QF)x_t\} \quad (3.93)$$

where

$$\text{ent} \equiv \beta \sum_{t=0}^{\infty} \beta^t w'_{t+1} w_{t+1}$$

The left side of inequality (3.93) is a straight line with slope  $\tilde{\theta}$

The upper bound on the left side of (3.93) is attained when

$$\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x'_t K(F, \tilde{\theta})' K(F, \tilde{\theta}) x_t \quad (3.94)$$

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 (3.92)
- Compute the maximizer  $V_{\tilde{\theta}}(x_0, F)$  and the associated entropy using (3.94)
- 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

$$\min_{\{u_t\}} \sum_{t=0}^{\infty} \beta^t \{ x'_t R x_t + u'_t Q u_t - \beta \theta w'_{t+1} w_{t+1} \} \quad (3.95)$$

where  $\{w_{t+1}\}$  satisfies  $w_{t+1} = Kx_t$

In other words, agent 1 minimizes

$$\sum_{t=0}^{\infty} \beta^t \{ x'_t (R - \beta \theta K' K) x_t + u'_t Q u_t \} \quad (3.96)$$

subject to

$$x_{t+1} = (A + CK)x_t + Bu_t \quad (3.97)$$

Once again, the expression for the optimal policy can be found *here* — we denote it by  $\tilde{F}$

**Nash Equilibrium** Clearly the  $\tilde{F}$  we have obtained depends on  $K$ , which, in agent 2's problem, depended on an initial policy  $F$

Holding all other parameters fixed, we can represent this relationship as a mapping  $\Phi$ , where

$$\tilde{F} = \Phi(K(F, \theta))$$

The map  $F \mapsto \Phi(K(F, \theta))$  corresponds to a situation in which

1. agent 1 uses an arbitrary initial policy  $F$

2. agent 2 best responds to agent 1 by choosing  $K(F, \theta)$
3. agent 1 best responds to agent 2 by choosing  $\tilde{F} = \Phi(K(F, \theta))$

As you may have already guessed, the robust policy  $\hat{F}$  defined in (3.83) is a fixed point of the mapping  $\Phi$

In particular, for any given  $\theta$ ,

1.  $K(\hat{F}, \theta) = \hat{K}$ , where  $\hat{K}$  is as given in (3.84)
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 (3.79) with the stochastic analogue

$$J(x) = \min_u \max_{\psi \in \mathcal{P}} \left\{ x' Rx + u' Qu + \beta \left[ \int J(Ax + Bu + Cw) \psi(dw) - \theta D_{KL}(\psi, \phi) \right] \right\} \quad (3.98)$$

Here  $\mathcal{P}$  represents the set of all densities on  $\mathbb{R}^n$  and  $\phi$  is the benchmark distribution  $N(0, I)$

The distribution  $\phi$  is chosen as the least desirable conditional distribution in terms of next period outcomes, while taking into account the penalty term  $\theta D_{KL}(\psi, \phi)$

This penalty term plays a role analogous to the one played by the deterministic penalty  $\theta w' w$  in (3.79), since it discourages large deviations from the benchmark

**Solving the Model** The maximization problem in (3.98) 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

$$\begin{aligned} \max_{\psi \in \mathcal{P}} & \left\{ \int (Ax + Bu + Cw)'P(Ax + Bu + Cw) \psi(dw) - \theta D_{KL}(\psi, \phi) \right\} \\ &= (Ax + Bu)'D(P)(Ax + Bu) + \kappa(\theta, P) \end{aligned} \quad (3.99)$$

where

$$\kappa(\theta, P) := \theta \ln[\det(I - \theta^{-1}C'PC)^{-1}]$$

and the maximizer is the Gaussian distribution

$$\psi = N\left((\theta I - C'PC)^{-1}C'P(Ax + Bu), (I - \theta^{-1}C'PC)^{-1}\right) \quad (3.100)$$

Substituting the expression for the maximum into Bellman equation (3.98) and using  $J(x) = x'Px + d$  gives

$$x'Px + d = \min_u \{x'Rx + u'Qu + \beta (Ax + Bu)'D(P)(Ax + Bu) + \beta [d + \kappa(\theta, P)]\} \quad (3.101)$$

Since constant terms do not affect minimizers, the solution is the same as (3.82), leading to

$$x'Px + d = x'\mathcal{B}(D(P))x + \beta [d + \kappa(\theta, P)]$$

To solve this Bellman equation, we take  $\hat{P}$  to be the positive definite fixed point of  $\mathcal{B} \circ \mathcal{D}$

In addition, we take  $\hat{d}$  as the real number solving  $d = \beta [d + \kappa(\theta, P)]$ , which is

$$\hat{d} := \frac{\beta}{1 - \beta} \kappa(\theta, P) \quad (3.102)$$

The robust policy in this stochastic case is the minimizer in (3.101), which is once again  $u = -\hat{F}x$  for  $\hat{F}$  given by (3.83)

Substituting the robust policy into (3.100) 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 (3.84)

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 (3.98), satisfies

$$J_F(x) = \max_{\psi \in \mathcal{P}} \left\{ x'(R + F'QF)x + \beta \left[ \int J_F((A - BF)x + Cw) \psi(dw) - \theta D_{KL}(\psi, \phi) \right] \right\}$$

Writing  $J_F(x) = x'P_Fx + d_F$  and applying the same argument used to derive (3.99) we get

$$x'P_Fx + d_F = x'(R + F'QF)x + \beta [x'(A - BF)'D(P_F)(A - BF)x + d_F + \kappa(\theta, P_F)]$$

To solve this we take  $P_F$  to be the fixed point

$$P_F = R + F'QF + \beta(A - BF)'D(P_F)(A - BF)$$

and

$$d_F := \frac{\beta}{1 - \beta} \kappa(\theta, P_F) = \frac{\beta}{1 - \beta} \theta \ln[\det(I - \theta^{-1}C'P_FC)^{-1}] \quad (3.103)$$

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

Here's the relevant code, from file `robustlq.py`

```
"""
Filename: robustlq.py

Authors: Chase Coleman, Spencer Lyon, Thomas Sargent, John Stachurski

Solves robust LQ control problems.

"""

from __future__ import division # Remove for Python 3.sx
from textwrap import dedent
import numpy as np
from .lqcontrol import LQ
from .quadsums import var_quadratic_sum
from numpy import dot, log, sqrt, identity, hstack, vstack, trace
from scipy.linalg import solve, inv, det
from .matrix_eqn import solve_discrete_lyapunov


class RBLQ(object):
    """
    Provides methods for analysing infinite horizon robust LQ control
    problems of the form

```

```

.. math::

    \min_{\{u_t\}} \sum_t \beta^t \{x_t' R x_t + u_t' Q u_t\}

subject to

.. math::

    x_{\{t+1\}} = A x_t + B u_t + C w_{\{t+1\}}
```

and with model misspecification parameter  $\theta$ .

*Parameters*

-----

- $Q$  : array\_like(float, ndim=2)  
The cost(payoff) matrix for the controls. See above for more.  
 $Q$  should be  $k \times k$  and symmetric and positive definite
- $R$  : array\_like(float, ndim=2)  
The cost(payoff) matrix for the state. See above for more.  $R$  should be  $n \times n$  and symmetric and non-negative definite
- $A$  : array\_like(float, ndim=2)  
The matrix that corresponds with the state in the state space system.  $A$  should be  $n \times n$
- $B$  : array\_like(float, ndim=2)  
The matrix that corresponds with the control in the state space system.  $B$  should be  $n \times k$
- $C$  : array\_like(float, ndim=2)  
The matrix that corresponds with the random process in the state space system.  $C$  should be  $n \times j$
- $\beta$  : scalar(float)  
The discount factor in the robust control problem
- $\theta$  : scalar(float)  
The robustness factor in the robust control problem

*Attributes*

-----

- $Q, R, A, B, C, \beta, \theta$  : see Parameters
- $k, n, j$  : scalar(int)  
The dimensions of the matrices

"""

```

def __init__(self, Q, R, A, B, C, beta, theta):

    # == Make sure all matrices can be treated as 2D arrays == #
    A, B, C, Q, R = list(map(np.atleast_2d, (A, B, C, Q, R)))
    self.A, self.B, self.C, self.Q, self.R = A, B, C, Q, R
    # == Record dimensions == #
    self.k = self.Q.shape[0]
    self.n = self.R.shape[0]
    self.j = self.C.shape[1]
    # == Remaining parameters == #
    self.beta, self.theta = beta, theta
```

```

def __repr__(self):
    return self.__str__()

def __str__(self):
    m = """\
Robust linear quadratic control system
- beta (discount parameter) : {b}
- theta (robustness factor) : {th}
- n (number of state variables) : {n}
- k (number of control variables) : {k}
- j (number of shocks) : {j}
"""
    return dedent(m.format(b=self.beta, n=self.n, k=self.k, j=self.j,
                           th=self.theta))

def d_operator(self, P):
    """
    The D operator, mapping P into

    .. math::

        D(P) := P + PC(\theta I - C'PC)^{-1}C'P.

    Parameters
    -----
    P : array_like(float, ndim=2)
        A matrix that should be n x n

    Returns
    -----
    dP : array_like(float, ndim=2)
        The matrix P after applying the D operator

    """
    C, theta = self.C, self.theta
    I = np.identity(self.j)
    S1 = dot(P, C)
    S2 = dot(C.T, S1)

    dP = P + dot(S1, solve(theta * I - S2, S1.T))

    return dP

def b_operator(self, P):
    """
    The B operator, mapping P into

    .. math::

        B(P) := R - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA + \beta A'PA

    and also returning

```

```

.. math::

    F := (Q + \beta B'PB)^{-1} \beta B'PA

Parameters
-----
P : array_like(float, ndim=2)
    A matrix that should be n x n

Returns
-----
F : array_like(float, ndim=2)
    The F matrix as defined above
new_P : array_like(float, ndim=2)
    The matrix P after applying the B operator

"""

A, B, Q, R, beta = self.A, self.B, self.Q, self.R, self.beta
S1 = Q + beta * dot(B.T, dot(P, B))
S2 = beta * dot(B.T, dot(P, A))
S3 = beta * dot(A.T, dot(P, A))
F = solve(S1, S2)
new_P = R - dot(S2.T, solve(S1, S2)) + S3

return F, new_P

def robust_rule(self):
    """
    This method solves the robust control problem by tricking it
    into a stacked LQ problem, as described in chapter 2 of Hansen-
    Sargent's text "Robustness." The optimal control with observed
    state is

    .. math::

        u_t = -F x_t

    And the value function is -x'Px

    Returns
    -----
    F : array_like(float, ndim=2)
        The optimal control matrix from above
    P : array_like(float, ndim=2)
        The positive semi-definite matrix defining the value
        function
    K : array_like(float, ndim=2)
        the worst-case shock matrix K, where
        :math:w_{t+1} = K x_t` is the worst case shock

    """

    # == Simplify names == #
    A, B, C, Q, R = self.A, self.B, self.C, self.Q, self.R

```

```

        beta, theta = self.beta, self.theta
        k, j = self.k, self.j
        # == Set up LQ version == #
        I = identity(j)
        Z = np.zeros((k, j))
        Ba = hstack([B, C])
        Qa = vstack([hstack([Q, Z]), hstack([Z.T, -beta*I*theta])])
        lq = LQ(Qa, R, A, Ba, beta=beta)

        # == Solve and convert back to robust problem == #
        P, f, d = lq.stationary_values()
        F = f[:, :]
        K = -f[k:f.shape[0], :]

    return F, K, P

def robust_rule_simple(self, P_init=None, max_iter=80, tol=1e-8):
    """
    A simple algorithm for computing the robust policy  $F$  and the corresponding value function  $P$ , based around straightforward iteration with the robust Bellman operator. This function is easier to understand but one or two orders of magnitude slower than self.robust_rule(). For more information see the docstring of that method.

    Parameters
    -----
    P_init : array_like(float, ndim=2), optional(default=None)
        The initial guess for the value function matrix. It will be a matrix of zeros if no guess is given
    max_iter : scalar(int), optional(default=80)
        The maximum number of iterations that are allowed
    tol : scalar(float), optional(default=1e-8)
        The tolerance for convergence

    Returns
    -----
    F : array_like(float, ndim=2)
        The optimal control matrix from above
    P : array_like(float, ndim=2)
        The positive semi-definite matrix defining the value function
    K : array_like(float, ndim=2)
        the worst-case shock matrix  $K$ , where :math:`w_{t+1} = K x_t` is the worst case shock

    """
    # == Simplify names == #
    A, B, C, Q, R = self.A, self.B, self.C, self.Q, self.R
    beta, theta = self.beta, self.theta
    # == Set up loop == #
    P = np.zeros((self.n, self.n)) if P_init is None else P_init
    iterate, e = 0, tol + 1

```

```

    while iterate < max_iter and e > tol:
        F, new_P = self.b_operator(self.d_operator(P))
        e = np.sqrt(np.sum((new_P - P)**2))
        iterate += 1
        P = new_P
    I = np.identity(self.j)
    S1 = P.dot(C)
    S2 = C.T.dot(S1)
    K = inv(theta * I - S2).dot(S1.T).dot(A - B.dot(F))

    return F, K, P

def F_to_K(self, F):
    """
    Compute agent 2's best cost-minimizing response K, given F.

    Parameters
    -----
    F : array_like(float, ndim=2)
        A k x n array

    Returns
    -----
    K : array_like(float, ndim=2)
        Agent's best cost minimizing response for a given F
    P : array_like(float, ndim=2)
        The value function for a given F

    """
    Q2 = self.beta * self.theta
    R2 = - self.R - dot(F.T, dot(self.Q, F))
    A2 = self.A - dot(self.B, F)
    B2 = self.C
    lq = LQ(Q2, R2, A2, B2, beta=self.beta)
    neg_P, neg_K, d = lq.stationary_values()

    return -neg_K, -neg_P

def K_to_F(self, K):
    """
    Compute agent 1's best value-maximizing response F, given K.

    Parameters
    -----
    K : array_like(float, ndim=2)
        A j x n array

    Returns
    -----
    F : array_like(float, ndim=2)
        The policy function for a given K
    P : array_like(float, ndim=2)
        The value function for a given K

```

```

    """
    A1 = self.A + dot(self.C, K)
    B1 = self.B
    Q1 = self.Q
    R1 = self.R - self.beta * self.theta * dot(K.T, K)
    lq = LQ(Q1, R1, A1, B1, beta=self.beta)
    P, F, d = lq.stationary_values()

    return F, P

def compute_deterministic_entropy(self, F, K, x0):
    """
    Given K and F, compute the value of deterministic entropy, which
    is sum_t beta^t x_t' K'K x_t with x_{t+1} = (A - BF + CK) x_t.

    Parameters
    -----
    F : array_like(float, ndim=2)
        The policy function, a k x n array
    K : array_like(float, ndim=2)
        The worst case matrix, a j x n array
    x0 : array_like(float, ndim=1)
        The initial condition for state

    Returns
    -----
    e : scalar(int)
        The deterministic entropy

    """
    H0 = dot(K.T, K)
    C0 = np.zeros((self.n, 1))
    A0 = self.A - dot(self.B, F) + dot(self.C, K)
    e = var_quadratic_sum(A0, C0, H0, self.beta, x0)

    return e

def evaluate_F(self, F):
    """
    Given a fixed policy F, with the interpretation u = -F x, this
    function computes the matrix P_F and constant d_F associated
    with discounted cost J_F(x) = x' P_F x + d_F.

    Parameters
    -----
    F : array_like(float, ndim=2)
        The policy function, a k x n array

    Returns
    -----
    P_F : array_like(float, ndim=2)
        Matrix for discounted cost

```

```

d_F : scalar(float)
    Constant for discounted cost
K_F : array_like(float, ndim=2)
    Worst case policy
O_F : array_like(float, ndim=2)
    Matrix for discounted entropy
o_F : scalar(float)
    Constant for discounted entropy

"""
# == Simplify names == #
Q, R, A, B, C = self.Q, self.R, self.A, self.B, self.C
beta, theta = self.beta, self.theta

# == Solve for policies and costs using agent 2's problem == #
K_F, P_F = self.F_to_K(F)
I = np.identity(self.j)
H = inv(I - C.T.dot(P_F.dot(C)) / theta)
d_F = log(det(H))

# == Compute O_F and o_F == #
sig = -1.0 / theta
A0 = sqrt(beta) * (A - dot(B, F) + dot(C, K_F))
O_F = solve_discrete_lyapunov(A0.T, beta * dot(K_F.T, K_F))
ho = (trace(H - 1) - d_F) / 2.0
tr = trace(dot(O_F, C.dot(H.dot(C.T))))
o_F = (ho + beta * tr) / (1 - beta)

return K_F, P_F, d_F, O_F, o_F

```

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 (3.83) – (3.84) 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 (3.89)
- `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

$$d_{t+1} = \rho d_t + \sigma_d w_{t+1}, \quad \{w_t\} \stackrel{\text{iid}}{\sim} N(0, 1)$$

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

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \rho \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ 0 \\ \sigma_d \end{bmatrix}$$

Our aim is to compute the value-entropy correspondences *shown above*

The parameters are

$$a_0 = 100, a_1 = 0.5, \rho = 0.9, \sigma_d = 0.05, \beta = 0.95, c = 2, \gamma = 50.0$$

The standard normal distribution for  $w_t$  is understood as the agent's baseline, with uncertainty parameterized by  $\theta$

We compute value-entropy correspondences for two policies

1. The no concern for robustness policy  $F_0$ , which is the ordinary LQ loss minimizer
2. A "moderate" concern for robustness policy  $F_b$ , with  $\theta = 0.02$

The code for producing the graph shown above, with blue being for the robust policy, is given in [robustness/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  $w_t \sim N(0, 1)$  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

$$x_t = (1, y_t, d_t) \text{ and } u_t = y_{t+1} - y_t$$

"""

```
import pandas as pd
import numpy as np
from scipy.linalg import eig
from scipy import interp
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(object):

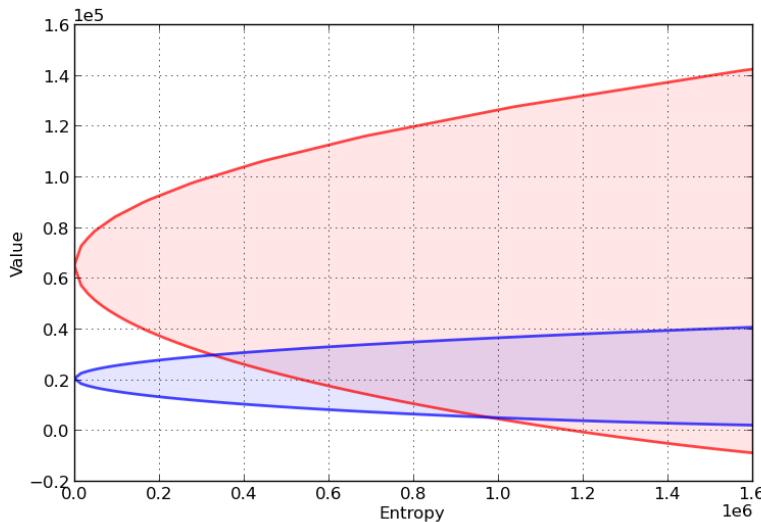
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __call__(self, z):
        return 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  $\theta$ ,  $K(\hat{F}, \theta) = \hat{K}$ , where  $\hat{K}$  is as given in (3.84)

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 (3.83), then

$$K(\hat{F}, \theta) = (\theta I - C' \hat{P} C)^{-1} C' \hat{P} (A - B \hat{F}) \quad (3.104)$$

*Proof:* As a first step, observe that when  $F = \hat{F}$ , the Bellman equation associated with the LQ problem (3.87) – (3.88) is

$$\tilde{P} = -R - \hat{F}' Q \hat{F} - \beta^2 (A - B \hat{F})' \tilde{P} C (\beta \theta I + \beta C' \tilde{P} C)^{-1} C' \tilde{P} (A - B \hat{F}) + \beta (A - B \hat{F})' \tilde{P} (A - B \hat{F}) \quad (3.105)$$

(revisit *this discussion* if you don't know where (3.105) 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 (3.105)

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 (3.104)

Hence it remains only to show that  $-\hat{P}$  solves (3.105), 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  $\hat{P} = \mathcal{B}(\mathcal{D}(\hat{P}))$ )

## 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

In problems that are recursive in the natural state variables, optimal decision rules are functions of the natural state variables

In this lecture, we describe an important class of problems that are not recursive in the natural state variables

Kydland and Prescott [KP77], [Pre77] and Calvo [Cal78] gave examples of such decision problems, which have the following features

- The time  $t \geq 0$  actions of some decision makers depend on the time  $s \geq t$  decisions of another decision maker called a government or Stackelberg leader
- At time  $t = 0$ , the government or Stackelberg leader chooses his actions for all times  $s \geq 0$
- In choosing actions for all times at time 0, the government or Stackelberg leader is said to *commit to a plan*

In these problems, variables that encode *history dependence* appear in optimal decision rules of the government or Stackelberg leader

Furthermore, the Stackelberg leader has distinct optimal decision rules for time  $t = 0$ , on the one hand, and times  $t \geq 1$ , on the other hand

The Stackelberg leader’s decision rules for  $t = 0$  and  $t \geq 1$  have distinct state variables

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 are not recursive in natural state variables

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

In such settings, private agents’ stocks of capital and other durable assets at time  $t$  are partly shaped by their past decisions that in turn were influenced by their earlier forecasts of the government’s actions

The *rational expectations* equilibrium concept plays an essential role

Rational expectations implies that in choosing its future actions, the government (or leader) chooses the private agents' (or followers') expectations about them

The government or leader understands and exploits that fact

In a rational expectations equilibrium, the government must confirm private agents' earlier forecasts of the government's time  $t$  actions

The requirement to confirm prior private sector forecasts puts constraints on the government's time  $t$  decisions that prevent its problem from being recursive in natural state variables

These additional constraints make the government'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 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 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 promises made earlier

Included in  $x_t$  might be prices and quantities that adjust instantaneously to clear markets at time  $t$

$$\text{Let } y_t = \begin{bmatrix} z_t \\ x_t \end{bmatrix}$$

Define the government's one-period loss function <sup>1</sup>

$$r(y, u) = y' R y + u' Q u \quad (3.106)$$

---

<sup>1</sup> 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).

Subject to an initial condition for  $z_0$ , but not for  $x_0$ , a government wants to maximize

$$-\sum_{t=0}^{\infty} \beta^t r(y_t, u_t) \quad (3.107)$$

The government makes policy in light of the model

$$\begin{bmatrix} I & 0 \\ G_{21} & G_{22} \end{bmatrix} \begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \hat{A}_{11} & \hat{A}_{12} \\ \hat{A}_{21} & \hat{A}_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + \hat{B}u_t \quad (3.108)$$

We assume that the matrix on the left is invertible, so that we can multiply both sides of the above equation by its inverse to obtain

$$\begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + Bu_t \quad (3.109)$$

or

$$y_{t+1} = Ay_t + Bu_t \quad (3.110)$$

The government maximizes (3.107) by choosing sequences  $\{u_t, x_t, z_{t+1}\}_{t=0}^{\infty}$  subject (3.110) and an initial condition for  $z_0$

Note that we have an intitial condition for  $z_0$  but not for  $x_0$

$x_0$  is among the variables to be chosen at time 0

The private sector's behavior is summarized by the second block of equations of (3.109) or (3.110)

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, these Euler equations summarize the private sector's best response to the sequence of actions by the government.

The government uses its understanding of these responses 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 (3.108) as

$$x_t = \phi_1 z_t + \phi_2 z_{t+1} + \phi_3 u_t + \phi_0 x_{t+1}, \quad (3.111)$$

where  $\phi_0 = \hat{A}_{22}^{-1} G_{22}$ .

The models we study in this chapter typically satisfy

*Forward-Looking Stability Condition* The eigenvalues of  $\phi_0$  are bounded in modulus by  $\beta^{-5}$ .

This stability condition makes equation (3.111) explosive if solved 'backwards' but stable if solved 'forwards'.

So we solve equation (3.111) forward to get

$$x_t = \sum_{j=0}^{\infty} \phi_0^j [\phi_1 z_{t+j} + \phi_2 z_{t+j+1} + \phi_3 u_{t+j}]. \quad (3.112)$$

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 (3.112).

The lecture on [history dependent policies](#) analyzes an example about *Ramsey taxation* in which, as is typical of such problems, the last  $n_x$  equations of (3.109) or (3.110) constitute *implementability constraints* that are formed by the Euler equations of a competitive fringe or private sector

When combined with a stability condition to be imposed below, these Euler equations summarize the private sector's best response to the sequence of actions by the government

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 (3.110), where  $\epsilon_{t+1}$  is an IID random vector with mean of zero and identity covariance matrix

Let  $X^t$  denote the history of any variable  $X$  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 (3.107) by choosing an  $x_0$  and a sequence of decision rules, the time  $t$  component of which maps the time  $t$  history of the natural state  $z^t$  into the 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<sup>2</sup> 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 (3.109)

### Solving the Stackelberg Problem

**Some Basic Notation** For any vector  $a_t$ , define  $\vec{a}_t = [a_t, a_{t+1}, \dots]$ .

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  $\Omega(y_0)$

---

<sup>2</sup> 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 Subproblems** Once again we use backward induction

We express the Stackelberg problem in terms of the following two subproblems:

**Subproblem 1**

$$v(y_0) = \max_{(\vec{y}_1, \vec{u}_0) \in \Omega(y_0)} - \sum_{t=0}^{\infty} \beta^t r(y_t, u_t) \quad (3.113)$$

**Subproblem 2**

$$w(z_0) = \max_{x_0} v(y_0) \quad (3.114)$$

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

**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

$$v(y) = \max_{u, y^*} \{-r(y, u) + \beta v(y^*)\} \quad (3.115)$$

where the maximization is subject to

$$y^* = Ay + Bu \quad (3.116)$$

and  $y^*$  denotes next period's value.

Substituting  $v(y) = -y'Py$  into Bellman equation (3.115) gives

$$-y'Py = \max_{u, y^*} \{-y'Ry - u'Qu - \beta y^{*\prime} Py^*\}$$

which as in lecture [linear regulator](#) gives rise to the algebraic matrix Riccati equation

$$P = R + \beta A'PA - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA \quad (3.117)$$

and the optimal decision rule coefficient vector

$$F = \beta(Q + \beta B'PB)^{-1}B'PA, \quad (3.118)$$

where the optimal decision rule is

$$u_t = -Fy_t. \quad (3.119)$$

**Subproblem 2** The value function  $v(y_0)$  satisfies

$$v(y_0) = -z'_0 P_{11} z_0 - 2x'_0 P_{21} z_0 - x'_0 P_{22} x_0 \quad (3.120)$$

where

$$P = \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix}$$

Now choose  $x_0$  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

$$x_0 = -P_{22}^{-1}P_{21}z_0. \quad (3.121)$$

**Summary** We solve the Stackelberg problem by

- formulating a particular optimal linear regulator,
- solving the associated matrix Riccati equation (3.117) for  $P$ ,
- computing  $F$ ,
- then partitioning  $P$  to obtain representation (3.121)

**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

This means that for  $t \geq 1$ ,  $x_t$  is *not* a linear function of  $z_t$  and that  $x_t$  exerts an independent influence on  $u_t$

The situation is different at  $t = 0$

For  $t = 0$ , the initialization the optimal choice of  $x_0 = -P_{22}^{-1}P_{21}z_0$  described in equation (3.121) implies that

$$u_0 = (f_{11} - f_{12}P_{22}^{-1}P_{21})z_0 \quad (3.122)$$

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 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  $\mu_x$  on the last  $n_x$  equations of (3.108) or (3.109)

These multipliers measure the cost today of honoring past government promises about current and future settings of  $u$

Later, we shall show that as a result of optimally choosing  $x_0$ , it is appropriate to initialize the multipliers to zero at time  $t = 0$

This is true because at  $t = 0$ , there are no past promises about  $u$  to honor

But the multipliers  $\mu_x$  take nonzero values thereafter, reflecting future costs to the government of adhering to its commitment

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  $\mu_{xt}$  on the forward-looking variables  $x_t$  evidently equals

$$\mu_{xt} = P_{21}z_t + P_{22}x_t. \quad (3.123)$$

So (3.121) is equivalent with

$$\mu_{x0} = 0. \quad (3.124)$$

**History-dependent representation of decision rule** For some purposes, it is useful to express the decision rule for  $u_t$  as a function of  $z_t$ ,  $z_{t-1}$ , and  $u_{t-1}$

This can be accomplished as follows

First note that  $u_t = -Fy_t$  implies that the *closed loop* law of motion for  $y$  is

$$y_{t+1} = (A - BF)y_t \quad (3.125)$$

which it is convenient to represent as

$$\begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} \quad (3.126)$$

and write the decision rule for  $u_t$

$$u_t = f_{11}z_t + f_{12}x_t \quad (3.127)$$

Then where  $f_{12}^{-1}$  denotes the generalized inverse of  $f_{12}$ , (3.127) implies  $x_t = f_{12}^{-1}(u_t - f_{11}z_t)$

Equate the right side of this expression to the right side of the second line of (3.126) lagged once and rearrange by using (3.127) lagged once and rearrange by using lagged once to eliminate  $\mu_{x,t-1}$  to get

$$u_t = f_{12}m_{22}f_{12}^{-1}u_{t-1} + f_{11}z_t + f_{12}(m_{21} - m_{22}f_{12}^{-1}f_{11})z_{t-1} \quad (3.128)$$

or

$$u_t = \rho u_{t-1} + \alpha_0 z_t + \alpha_1 z_{t-1} \quad (3.129)$$

for  $t \geq 1$

By making the instrument feed back on itself, the form of decision rule (3.129) potentially allows for “instrument-smoothing” to emerge as an optimal rule under commitment

### 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<sup>3</sup>

The costs of production are  $C_t = eQ_t + .5gQ_t^2 + .5c(Q_{t+1} - Q_t)^2$  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

$$p_t = A_0 - A_1(Q_t + \bar{q}_t) + v_t, \quad (3.130)$$

where  $A_0, A_1$  are both positive and  $v_t$  is a disturbance to demand governed by

$$v_{t+1} = \rho v_t + C_\epsilon \check{\epsilon}_{t+1} \quad (3.131)$$

and where  $|\rho| < 1$  and  $\check{\epsilon}_{t+1}$  is an IID sequence of random variables with mean zero and variance 1

In (3.130),  $\bar{q}_t$  is equilibrium output of the representative competitive firm

In equilibrium,  $\bar{q}_t = q_t$ , but we must distinguish between  $q_t$  and  $\bar{q}_t$  in posing the optimum problem of a competitive firm

**The competitive fringe** The representative competitive firm regards  $\{p_t\}_{t=0}^\infty$  as an exogenous stochastic process and chooses an output plan to maximize

$$E_0 \sum_{t=0}^{\infty} \beta^t \{p_t q_t - \sigma_t\}, \quad \beta \in (0, 1) \quad (3.132)$$

subject to  $q_0$  given, where  $E_t$  is the mathematical expectation based on time  $t$  information

---

<sup>3</sup> [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.

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 (3.132) are

$$i_t = E_t \beta i_{t+1} - c^{-1} \beta h q_{t+1} + c^{-1} \beta E_t (p_{t+1} - d) \quad (3.133)$$

for  $t \geq 0$

We appeal to a *certainty equivalence principle* to justify working with a non-stochastic version of (3.133) formed by dropping the expectation operator and the random term  $\varepsilon_{t+1}$  from (3.131)

We use a method of [Sar79] and [Tow83]<sup>4</sup>

We shift (3.130) forward one period, replace conditional expectations with realized values, use (3.130) to substitute for  $p_{t+1}$  in (3.133), and set  $q_t = \bar{q}_t$  and  $i_t = \bar{i}_t$  for all  $t \geq 0$  to get

$$\bar{i}_t = \beta \bar{i}_{t+1} - c^{-1} \beta h \bar{q}_{t+1} + c^{-1} \beta (A_0 - d) - c^{-1} \beta A_1 \bar{q}_{t+1} - c^{-1} \beta A_1 Q_{t+1} + c^{-1} \beta v_{t+1} \quad (3.134)$$

Given sufficiently stable sequences  $\{Q_t, v_t\}$ , we could solve (3.134) and  $\bar{i}_t = \bar{q}_{t+1} - \bar{q}_t$  to express the competitive fringe's output sequence as a function of the (tail of the) monopolist's output sequence

The dependence of  $\bar{i}_t$  on future  $Q_t$ 's opens an avenue for the monopolist to influence current outcomes by its future actions

It is this feature that makes the monopolist's problem fail to be recursive in the natural state variables  $\bar{q}, Q$

The monopolist arrives at period  $t > 0$  facing the constraint that it must confirm the expectations about its time  $t$  decision upon which the competitive fringe based its decisions at dates before  $t$

**The 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:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ A_0 - d & 1 & -A_1 & -A_1 - h & c \end{bmatrix} \begin{bmatrix} 1 \\ v_{t+1} \\ Q_{t+1} \\ \bar{q}_{t+1} \\ \bar{i}_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \rho & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & \frac{c}{\beta} \end{bmatrix} \begin{bmatrix} 1 \\ v_t \\ Q_t \\ \bar{q}_t \\ \bar{i}_t \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} u_t \quad (3.135)$$

where  $u_t = Q_{t+1} - Q_t$  is the control of the monopolist at time  $t$

The last row portrays the implementability constraints (3.134)

<sup>4</sup> 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.

Represent (3.135) as

$$y_{t+1} = Ay_t + Bu_t \quad (3.136)$$

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 (3.136),  $\bar{i}_t$  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 (3.136) as the transition law

Let's decode it

To match our general setup, we partition  $y_t$  as  $y'_t = [z'_t \ x'_t]$  where  $z'_t = [1 \ v_t \ Q_t \ \bar{q}_t]$  and  $x_t = \bar{i}_t$

The monopolist's problem is

$$\max_{\{u_t, p_{t+1}, Q_{t+1}, \bar{q}_{t+1}, \bar{i}_t\}} \sum_{t=0}^{\infty} \beta^t \{p_t Q_t - C_t\}$$

subject to the given initial condition for  $z_0$ , equations (3.130) and (3.134) and  $\bar{i}_t = \bar{q}_{t+1} - \bar{q}_t$ , 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 (3.130) and the implementability constraint (3.134) that describes the best responses of firms in the competitive fringe

By substituting (3.130) into the above objective function, the monopolist's problem can be expressed as

$$\max_{\{u_t\}} \sum_{t=0}^{\infty} \beta^t \{(A_0 - A_1(\bar{q}_t + Q_t) + v_t)Q_t - eQ_t - .5gQ_t^2 - .5cu_t^2\} \quad (3.137)$$

subject to (3.136)

This can be written

$$\max_{\{u_t\}} - \sum_{t=0}^{\infty} \beta^t \{y'_t R y_t + u'_t Q u_t\} \quad (3.138)$$

subject to (3.136) 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

$$\bar{y}_{t+1} = (A - BF)\bar{y}_t \quad (3.139)$$

where  $\bar{y}_t = [1 \ v_t \ Q_t \ \bar{q}_t \ \bar{i}_t]'$

**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

$$p_t = a_0 - a_1 Q_t - a_1 \bar{q}_t + v_t \quad (3.140)$$

$$\equiv E_p [\bar{y}_t] \quad (3.141)$$

(Please remember that  $\bar{q}_t$  is a component of  $\bar{y}_t$ )

From the point of view of a representative firm in the competitive fringe,  $\{\bar{y}_t\}$  is an exogenous process

A representative fringe firm wants to forecast  $\bar{y}$  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  $\bar{q}$

An individual follower firm confronts state  $[\bar{y}_t \ q_t]'$  where  $q_t$  is its current output as opposed to  $\bar{q}$  within  $\bar{y}$

It believes that it chooses future values of  $q_t$  but not future values of  $\bar{q}_t$

(This is an application of a “Big  $K$ , little  $k$ ” idea)

The follower faces law of motion

$$\begin{bmatrix} \bar{y}_{t+1} \\ q_{t+1} \end{bmatrix} = \begin{bmatrix} A - BF & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \bar{y}_t \\ q_t \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} i_t \quad (3.142)$$

We calculated  $F$  and therefore  $A - BF$  earlier

We can restate the optimization problem of the representative competitive firm

The firm takes  $\bar{y}_t$  as an exogenous process and chooses an output plan  $\{q_t\}$  to maximize

$$E_0 \sum_{t=0}^{\infty} \beta^t \{ p_t q_t - \sigma_t \}, \quad \beta \in (0, 1)$$

subject to  $q_0$  given the law of motion (3.139) and the price function (3.140) and where the costs are still  $\sigma_t = d q_t + .5 h q_t^2 + .5 c (q_{t+1} - q_t)^2$

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

$$i_t = -F_s \begin{bmatrix} 1 \\ v_t \\ Q_t \\ \bar{q}_t \\ \dot{i}_t \\ q_t \end{bmatrix} \quad (3.143)$$

Now let's stare at the decision rule (3.143) 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 (3.143) to have the property that  $i_t = \bar{i}_t$  when we evaluate it at  $q_t = \bar{q}_t$

We inherit these desires from a "Big  $K$ , little  $k$ " logic

Here we apply a "Big  $K$ , little  $k$ " logic in two parts to make the "representative firm be representative" *after* solving the representative firm's optimization problem

- We want  $q_t = \bar{q}_t$
- We want  $i_t = \bar{i}_t$

**Numerical example** We computed the optimal Stackelberg plan for parameter settings  $A_0, A_1, \rho, C_c, c, d, e, g, h, \beta = 100, 1, .8, .2, 1, 20, 20, .2, .95$ <sup>5</sup>

For these parameter values the monopolist's decision rule is

$$u_t = (Q_{t+1} - Q_t) = [83.98 \quad 0.78 \quad -0.95 \quad -1.31 \quad -2.07] \begin{bmatrix} 1 \\ v_t \\ Q_t \\ \bar{q}_t \\ \bar{i}_t \end{bmatrix}$$

for  $t \geq 0$

and

$$x_0 \equiv \bar{i}_0 = [31.08 \quad 0.29 \quad -0.15 \quad -0.56] \begin{bmatrix} 1 \\ v_0 \\ Q_0 \\ \bar{q}_0 \end{bmatrix}$$

For this example, starting from  $z_0 = [1 \quad v_0 \quad Q_0 \quad \bar{q}_0] = [1 \quad 0 \quad 25 \quad 46]$ , the monopolist chooses to set  $i_0 = 1.43$

That choice implies that

- $i_1 = 0.25$ , and
- $z_1 = [1 \quad v_1 \quad Q_1 \quad \bar{q}_1] = [1 \quad 0 \quad 21.83 \quad 47.43]$

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

---

<sup>5</sup> These calculations were performed by the Python program from QuantEcon.applications in dyn\_stack/oligopoly.py.

The recursive representation of the decision rule for a representative fringe firm is

$$i_t = [0 \ 0 \ 0 \ .34 \ 1 \ -.34] \begin{bmatrix} 1 \\ v_t \\ Q_t \\ \bar{q}_t \\ \dot{i}_t \\ q_t \end{bmatrix},$$

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<sup>6</sup>

We shall encounter another example in the lecture on history dependent policies

There are many more such problems - see chapters 20-24 of [LS12]

### Exercises

**Exercise 1** There is no uncertainty

For  $t \geq 0$ , a monetary authority sets the growth of (the log of) money according to

$$m_{t+1} = m_t + u_t \quad (3.144)$$

subject to the initial condition  $m_0 > 0$  given

The demand for money is

$$m_t - p_t = -\alpha(p_{t+1} - p_t) \quad (3.145)$$

where  $\alpha > 0$  and  $p_t$  is the log of the price level

Equation (3.144) can be interpreted as the Euler equation of the holders of money

- a. Briefly interpret how (3.144) makes the demand for real balances vary inversely with the expected rate of inflation. Temporarily (only for this part of the exercise) drop (3.144) and assume instead that  $\{m_t\}$  is a given sequence satisfying  $\sum_{t=0}^{\infty} m_t^2 < +\infty$ . Solve the difference equation (3.144) "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  $\{u_t\}_{t=0}^{\infty}$

<sup>6</sup> For another application of the techniques in this lecture and how they related to the method recommended by [KP80b], please see [this lecture](#).

The monetary authority orders sequences  $\{m_t, p_t\}_{t=0}^{\infty}$  according to

$$-\sum_{t=0}^{\infty} .95^t [(p_t - \bar{p})^2 + u_t^2 + .00001m_t^2] \quad (3.146)$$

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  $.00001m_t^2$  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  $t = 0, \dots, 10$

*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

$$\sum_{t=0}^{\infty} \beta^t \{- .5(b - c_t)^2\} \quad (3.147)$$

where  $\beta \in (0, 1)$ ,  $b = 30$ , and  $c_t$  is time  $t$  consumption

The consumer faces a sequence of budget constraints

$$c_t + a_{t+1} = (1 + r)a_t + y_t - \tau_t \quad (3.148)$$

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

$$y_t = \rho y_{t-1}, \quad t \geq 1, \quad (3.149)$$

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

$$\sum_{t=0}^{\infty} \beta^t \{- .5(c_t - b)^2 - \tau_t^2\} \quad (3.150)$$

over  $\{c_t, \tau_t\}_{t=0}^{\infty}$  subject to the implementability constraints in (3.148) for  $t \geq 0$  and

$$\lambda_t = \beta(1+r)\lambda_{t+1} \quad (3.151)$$

for  $t \geq 0$ , where  $\lambda_t \equiv (b - c_t)$

- a. Argue that (3.151) is the Euler equation for a consumer who maximizes (3.147) subject to (3.148), 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  $\tau_t$

## Optimal Taxation

### Contents

- *Optimal Taxation*
  - *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

$$-\mathbb{E} \frac{1}{2} \sum_{t=0}^{\infty} \beta^t [(c_t - b_t)^2 + \ell_t^2] \quad (3.152)$$

subject to the budget constraint

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t p_t^0 [d_t + (1 - \tau_t) \ell_t + s_t - c_t] = 0 \quad (3.153)$$

Here

- $\beta$  is a discount factor in  $(0, 1)$
- $p_t^0$  is state price at time  $t$

- $b_t$  is a stochastic preference parameter
- $d_t$  is an endowment process
- $\tau_t$  is a flat tax rate on labor income
- $s_t$  is a promised time- $t$  coupon payment on debt issued by the government

The budget constraint 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 parameter  $b_t$  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

$$c_t + g_t = d_t + \ell_t \quad (3.154)$$

A labor-consumption process  $\{\ell_t, c_t\}$  is called *feasible* if (3.154) holds for all  $t$

**Government budget constraint** Where  $p_t^0$  is a scaled Arrow-Debreu price, the time zero government budget constraint is

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t p_t^0 (s_t + g_t - \tau_t \ell_t) = 0 \quad (3.155)$$

**Equilibrium** An *equilibrium* is a feasible allocation  $\{\ell_t, c_t\}$ , a sequence of prices  $\{p_t\}$ , and a tax system  $\{\tau_t\}$  such that

1. The allocation  $\{\ell_t, c_t\}$  is optimal for the household given  $\{p_t\}$  and  $\{\tau_t\}$
2. The government's budget constraint (3.155) is satisfied

The *Ramsey problem* is to choose the equilibrium  $\{\ell_t, c_t, \tau_t, p_t\}$  that maximizes the household's welfare

If  $\{\ell_t, c_t, \tau_t, p_t\}$  is a solution to 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 (3.155) in terms of exogenous variables and allocations
3. Maximize the household's objective function (3.152) subject to the constraint constructed in step 2 and the feasibility constraint (3.154)

The solution to this maximization problem pins down all quantities of interest

**Solution** Step one is to obtain the first order conditions for the household's problem, taking taxes and prices as given

Letting  $\mu$  be the Lagrange multiplier on (3.153), the first order conditions are  $p_t = (c_t - b_t)/\mu$  and  $\ell_t = (c_t - b_t)(1 - \tau_t)$

Rearranging and normalizing at  $\mu = b_0 - c_0$ , we can write these conditions as

$$p_t = \frac{b_t - c_t}{b_0 - c_0} \quad \text{and} \quad \tau_t = 1 - \frac{\ell_t}{b_t - c_t} \quad (3.156)$$

Substituting (3.156) into the government's budget constraint (3.155) yields

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t [(b_t - c_t)(s_t + g_t - \ell_t) + \ell_t^2] = 0 \quad (3.157)$$

The Ramsey problem now amounts to maximizing (3.152) subject to (3.157) and (3.154)

The associated Lagrangian is

$$\mathcal{L} = \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left\{ -\frac{1}{2} [(c_t - b_t)^2 + \ell_t^2] + \lambda [(b_t - c_t)(\ell_t - s_t - g_t) - \ell_t^2] + \mu_t [d_t + \ell_t - c_t - g_t] \right\} \quad (3.158)$$

The first order conditions associated with  $c_t$  and  $\ell_t$  are

$$-(c_t - b_t) + \lambda[-\ell_t + (g_t + s_t)] = \mu_t$$

and

$$\ell_t - \lambda[(b_t - c_t) - 2\ell_t] = \mu_t$$

Combining these last two equalities with (3.154) and working through the algebra, one can show that

$$\ell_t = \bar{\ell}_t - \nu m_t \quad \text{and} \quad c_t = \bar{c}_t - \nu m_t \quad (3.159)$$

where

- $\nu := \lambda / (1 + 2\lambda)$
- $\bar{\ell}_t := (b_t - d_t + g_t) / 2$
- $\bar{c}_t := (b_t + d_t - g_t) / 2$
- $m_t := (b_t - d_t - s_t) / 2$

Apart from  $\nu$ , all of these quantities are expressed in terms of exogenous variables

To solve for  $\nu$ , we can use the government's budget constraint again

The term inside the brackets in (3.157) is  $(b_t - c_t)(s_t + g_t) - (b_t - c_t)\ell_t + \ell_t^2$

Using (3.159), 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 (3.157), we get

$$\mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t (b_t - \bar{c}_t)(g_t + s_t) \right\} + (\nu^2 - \nu) \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t 2m_t^2 \right\} = 0 \quad (3.160)$$

Although it might not be clear yet, we are nearly there:

- The two expectations terms in (3.160) can be solved for in terms of model primitives
- This in turn allows us to solve for the Lagrange multiplier  $\nu$
- With  $\nu$  in hand, we can go back and solve for the allocations via (3.159)
- Once we have the allocations, prices and the tax system can be derived from (3.156)

**Solving the Quadratic Term** Let's consider how to obtain the term  $\nu$  in (3.160)

If we can solve the two expected geometric sums

$$b_0 := \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t (b_t - \bar{c}_t)(g_t + s_t) \right\} \quad \text{and} \quad a_0 := \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t 2m_t^2 \right\} \quad (3.161)$$

then the problem reduces to solving

$$b_0 + a_0(\nu^2 - \nu) = 0$$

for  $\nu$

Provided that  $4b_0 < a_0$ , there is a unique solution  $\nu \in (0, 1/2)$ , and a unique corresponding  $\lambda > 0$

Let's work out how to solve the expectations terms in (3.161)

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 (3.161), 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 of these expectations terms are special cases of the expression

$$q(x_0) = \mathbb{E} \sum_{t=0}^{\infty} \beta^t x'_t H x_t \quad (3.162)$$

where  $H$  is a conformable matrix, and  $x'_t$  is the transpose of column vector  $x_t$

Suppose first that  $\{x_t\}$  is the Gaussian VAR described *above*

In this case, the formula for computing  $q(x_0)$  is known to be  $q(x_0) = x'_0 Q x_0 + v$ , where

- $Q$  is the solution to  $Q = H + \beta A' Q A$ , and
- $v = \text{trace}(C' Q C) \beta / (1 - \beta)$

The first equation is known as a discrete Lyapunov equation, and can be solved using [this function](#)

Next suppose that  $\{x_t\}$  is the discrete Markov process described *above*

Suppose further that each  $x_t$  takes values in the state space  $\{x^1, \dots, x^N\} \subset \mathbb{R}^k$

Let  $h: \mathbb{R}^k \rightarrow \mathbb{R}$  be a given function, and suppose that we wish to evaluate

$$q(x_0) = \mathbb{E} \sum_{t=0}^{\infty} \beta^t h(x_t) \quad \text{given } x_0 = x^j$$

For example, in the discussion above,  $h(x_t) = x'_t H x_t$

It is legitimate to pass the expectation through the sum, leading to

$$q(x_0) = \sum_{t=0}^{\infty} \beta^t (P^t h)[j] \quad (3.163)$$

Here

- $P^t$  is the  $t$ -th power of the transition matrix  $P$
- $h$  is, with some abuse of notation, the vector  $(h(x^1), \dots, h(x^N))$
- $(P^t h)[j]$  indicates the  $j$ -th element of  $P^t h$

It can be show that (3.163) 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

One is the present value of government obligations outstanding at time  $t$ , which can be expressed as

$$B_t := \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j p_{t+j}^t (\tau_{t+j} \ell_{t+j} - g_{t+j}) \quad (3.164)$$

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 variation is more convenient for computation

Yet another way to write  $B_t$  is

$$B_t = \sum_{j=0}^{\infty} R_{tj}^{-1} (\tau_{t+j} \ell_{t+j} - g_{t+j})$$

where

$$R_{tj}^{-1} := \mathbb{E}_t \beta^j p_{t+j}^t$$

Here  $R_{tj}$  can be thought of as the gross  $j$ -period risk-free rate on holding government debt between  $t$  and  $j$

Furthermore, letting  $R_t$  be the one-period risk-free rate, we define

$$\pi_{t+1} := B_{t+1} - R_t [B_t - (\tau_t \ell_t - g_t)]$$

and

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

The term  $\pi_{t+1}$  is the payout on the public's portfolio of government debt

As shown in the original manuscript, if we distort one-step-ahead transition probabilities by the adjustment factor

$$\xi_t := \frac{p_{t+1}^t}{\mathbb{E}_t p_{t+1}^t}$$

then  $\Pi_t$  is a martingale under the distorted probabilities

See the treatment in the manuscript for more discussion and intuition

For now we will concern ourselves with computation

### 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/lqramsey.py` from the `applications` repository

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
        run        - 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 (3.162) 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 function `gen_fig_1()`

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/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 = lqramsey.Economy(beta=beta,
                            Sg=Sg,
                            Sd=Sd,
                            Sb=Sb,
                            Ss=Ss,
```

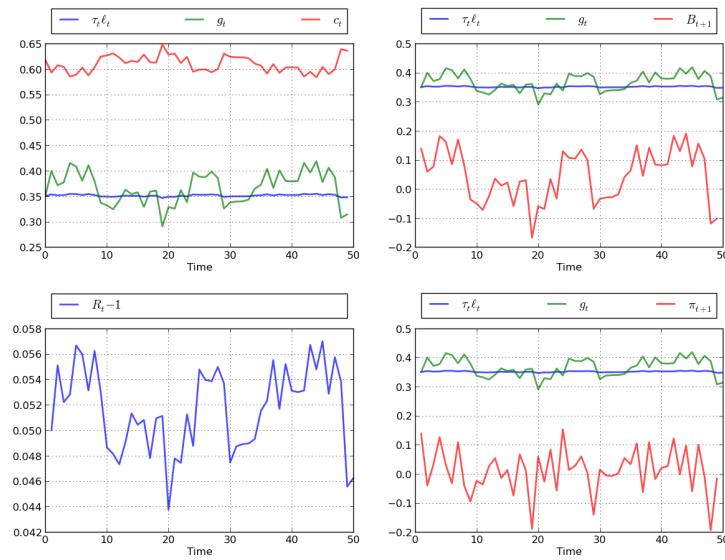
```

discrete=False,
proc=(A, C))

T = 50
path = lqramsey.compute_paths(T, economy)
lqramsey.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/lqramsey_discrete.py`

```

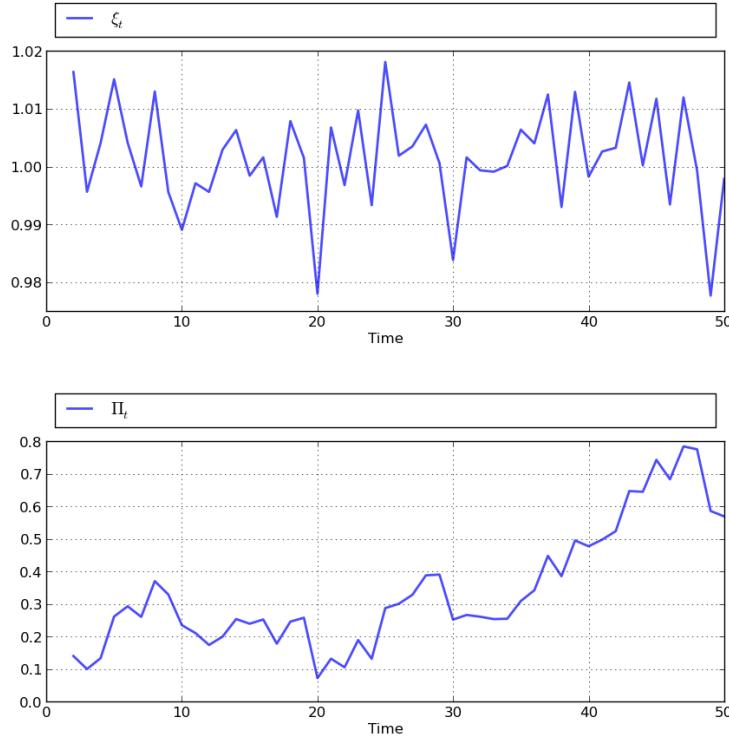
"""
Filename: lqramsey_discrete.py
Authors: Thomas Sargent, Doc-Jin Jang, Jeong-hun Choi, John Stachurski

LQ Ramsey model with discrete exogenous process.

"""

from numpy import array
import lqramsey

```



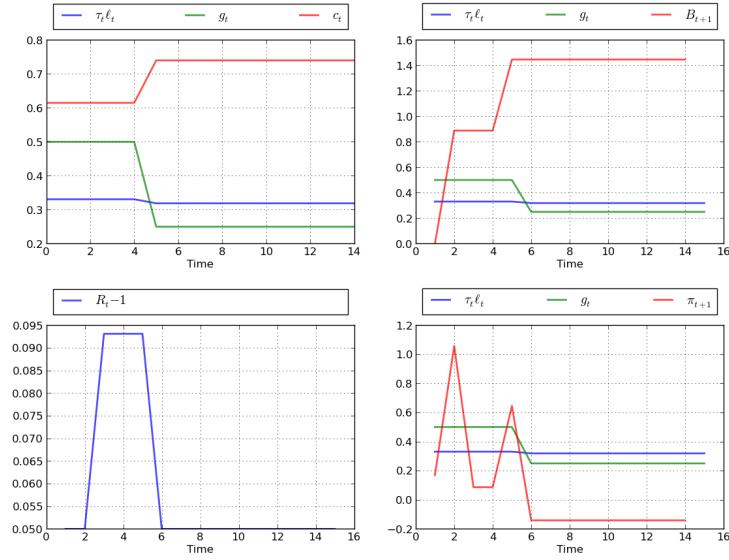
```
# == Parameters ==
beta = 1 / 1.05
P = array([[0.8, 0.2, 0.0],
           [0.0, 0.5, 0.5],
           [0.0, 0.0, 1.0]])
# == Possible states of the world ==
# Each column is a state of the world. The rows are [g d b s 1]
x_vals = array([[0.5, 0.5, 0.25],
                [0.0, 0.0, 0.0],
                [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 = lqramsey.Economy(beta=beta,
                            Sg=Sg,
                            Sd=Sd,
                            Sb=Sb,
                            Ss=Ss,
                            discrete=True,
                            proc=(P, x_vals))

T = 15
```

```
path = lqramsey.compute_paths(T, economy)
lqramsey.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

$$g_{t+1} - \mu_g = \rho(g_{t-3} - \mu_g) + C_g w_{g,t+1}$$

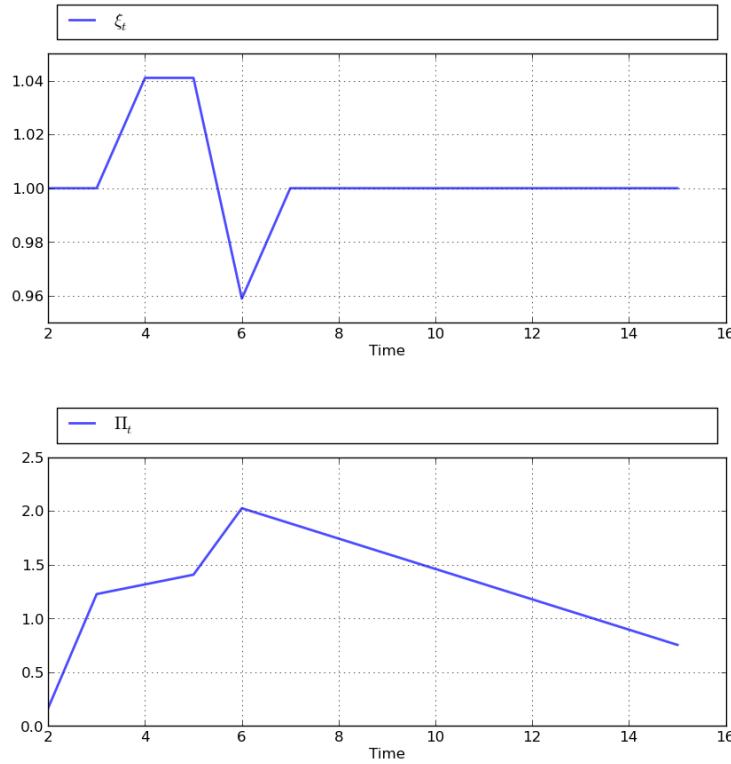
with  $\rho = 0.95$  and  $C_g = 0.7\sqrt{1-\rho^2}$

Produce the corresponding figures

### Solutions

[Solution notebook](#)

## 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<sup>1</sup>

**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](#)

## 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

---

<sup>1</sup> We could also call a competitive equilibrium a rational expectations equilibrium.

**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

$$p_t = A_0 - A_1 Q_t, \quad A_0 > 0, A_1 > 0 \quad (3.165)$$

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  $\tau_t$  per unit of output
- treats  $\{p_t, \tau_t\}_{t=0}^{\infty}$  as exogenous
- chooses  $\{q_{t+1}\}_{t=0}^{\infty}$  to maximize

$$\sum_{t=0}^{\infty} \beta^t \left\{ p_t q_t - \frac{d}{2}(q_{t+1} - q_t)^2 - \tau_t q_t \right\} \quad (3.166)$$

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

$$u_t = \frac{\beta}{d} p_{t+1} + \beta u_{t+1} - \frac{\beta}{d} \tau_{t+1}, \quad t = 0, 1, \dots \quad (3.167)$$

To compute a competitive equilibrium, it is appropriate to take (3.167), eliminate  $p_t$  in favor of  $Q_t$  by using (3.165), and then set  $q_t = Q_t$

This last step *makes the representative firm be representative*<sup>2</sup>

We arrive at

$$u_t = \frac{\beta}{d} (A_0 - A_1 Q_{t+1}) + \beta u_{t+1} - \frac{\beta}{d} \tau_{t+1} \quad (3.168)$$

$$Q_{t+1} = Q_t + u_t \quad (3.169)$$

**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 (3.165), (3.168), and (3.169)

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:

- A continuation  $\vec{\tau}_1 = \{\tau_t\}_{t=1}^\infty$  of a tax policy  $\vec{\tau}$  influences  $u_0$  via (3.168) entirely through its impact on  $u_1$

A continuation competitive equilibrium can be indexed by a  $u_1$  that satisfies (3.168)

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<sup>3</sup>

Define  $Q^t := [Q_0, \dots, Q_t]$

A **history-dependent tax policy** is a sequence of functions  $\{\sigma_t\}_{t=0}^\infty$  with  $\sigma_t$  mapping  $Q^t$  into a choice of  $\tau_{t+1}$

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

<sup>2</sup> 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.

<sup>3</sup> 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  $d_{u_t}$  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  $(Q_{t+1}, \tau_{t+1})$ , 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  $u_t$ .

### 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

$$A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 \quad (3.170)$$

At time  $t = 0$ , a Ramsey planner faces the intertemporal budget constraint

$$\sum_{t=1}^{\infty} \beta^t \tau_t Q_t = G_0 \quad (3.171)$$

Note that (3.171) 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

$$\sum_{t=0}^{\infty} \beta^t \left\{ A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 \right\} \quad (3.172)$$

subject to (3.171)

Thus, the Ramsey timing protocol is:

1. At time 0, knowing  $(Q_0, G_0)$ , the Ramsey planner chooses  $\{\tau_{t+1}\}_{t=0}^{\infty}$
2. Given  $(Q_0, \{\tau_{t+1}\}_{t=0}^{\infty})$ , 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 (3.172) subject to (3.168), (3.169), and (3.171)

To formulate this problem as a Lagrangian, attach a Lagrange multiplier  $\mu$  to the budget constraint (3.171)

Then the planner chooses  $\{u_t\}_{t=0}^{\infty}$ ,  $\{\tau_t\}_{t=1}^{\infty}$  to maximize and the Lagrange multiplier  $\mu$  to minimize

$$\sum_{t=0}^{\infty} \beta^t (A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2) + \mu \left[ \sum_{t=0}^{\infty} \beta^t \tau_t Q_t - G_0 - \tau_0 Q_0 \right] \quad (3.173)$$

subject to and (3.168) and (3.169)

The Ramsey problem is a special case of the linear quadratic dynamic Stackelberg problem analyzed in [this lecture](#)

The key implementability conditions are (3.168) for  $t \geq 0$

Holding fixed  $\mu$  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  $\tau_t$  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  $\tau_{t+1}$  will not depend on  $\tau_t$

The government chooses  $\tau_{t+1}$  at time  $t$  as a function of the time  $t$  state

Thus, we can rewrite the Ramsey problem as

$$\max_{\{y_t, \tau_{t+1}\}} - \sum_{t=0}^{\infty} \beta^t y_t' R y_t \quad (3.174)$$

subject to  $z_0$  given and the law of motion

$$y_{t+1} = A y_t + B \tau_{t+1} \quad (3.175)$$

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  $Q_t, \tau_t, u_t$ .

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  $z_t = [1 \ Q_t \ \tau_t]'$  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

$$v(y_t) = \max_{\tau_{t+1}} \{-y_t' R y_t + \beta v(y_{t+1})\} \quad (3.176)$$

where the maximization is subject to the constraint

$$y_{t+1} = A y_t + B \tau_{t+1}$$

and where

$$R = \begin{bmatrix} 0 & -\frac{A_0}{2} & 0 & 0 \\ -\frac{A_0}{2} & \frac{A_1}{2} & -\frac{\mu}{2} & 0 \\ 0 & \frac{-\mu}{2} & 0 & 0 \\ 0 & 0 & 0 & \frac{d}{2} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ -\frac{A_0}{d} & \frac{A_1}{d} & 0 & \frac{A_1}{d} + \frac{1}{\beta} \end{bmatrix}, \text{ and } B = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \frac{1}{d} \end{bmatrix}.$$

Functional equation (3.176) has solution

$$v(y_t) = -y_t' P y_t$$

where

- $P$  solves the algebraic matrix Riccati equation  $P = R + \beta A'PA - \beta A'PB(B'PB)^{-1}B'PA$
- the optimal policy function is given by  $\tau_{t+1} = -Fy_t$  for  $F = (B'PB)^{-1}B'PA$

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

$$0 = \frac{\partial}{\partial u_0} (z_0' P_{11} z_0 + z_0' P_{12} u_0 + u_0' P_{21} z_0 + u_0' P_{22} u_0) = P_{12}' z_0 + P_{21} z_0 + 2P_{22} u_0$$

which implies

$$u_0 = -P_{22}^{-1} P_{21} z_0 \quad (3.177)$$

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  $[z_0 \ -P_{22}^{-1} P_{21} z_0]'$

**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

$$u_0 = v(Q_0 | \mu) \quad (3.178)$$

that obeys (3.177) and the following updating equations for  $t \geq 0$ :

$$\tau_{t+1} = \tau(Q_t, u_t | \mu) \quad (3.179)$$

$$Q_{t+1} = Q_t + u_t \quad (3.180)$$

$$u_{t+1} = u(Q_t, u_t | \mu) \quad (3.181)$$

We have conditioned the functions  $v$ ,  $\tau$ , and  $u$  by  $\mu$  to emphasize how the dependence of  $F$  on  $G_0$  appears indirectly through the Lagrange multiplier  $\mu$

**An Example Calculation** We'll discuss how to compute  $\mu$  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 matrixies
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]

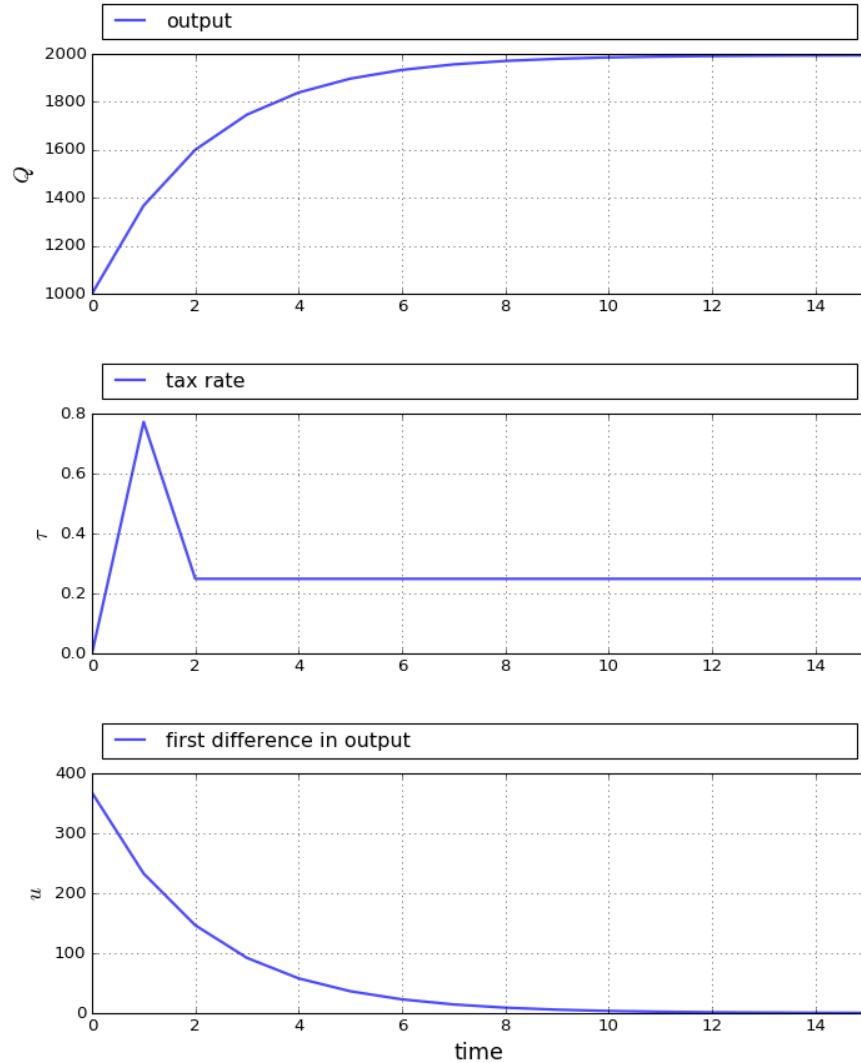
if __name__ == '__main__':
    print("1 Q tau u")
    print(y)
    print("-F")
    print(-F)

```

The program can also be found in the QuantEcon GitHub repository

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  $\tau$  and the Ramsey out-



come for  $(Q_t, u_t)$

From top to bottom, the panels show  $Q_t$ ,  $\tau_t$  and  $u_t := Q_{t+1} - Q_t$  over  $t = 0, \dots, 15$

The optimal decision rule is <sup>4</sup>

$$\tau_{t+1} = -248.0624 - 0.1242Q_t - 0.3347u_t \quad (3.182)$$

Notice how the Ramsey plan calls for a high tax at  $t = 1$  followed by a perpetual stream of lower taxes

---

<sup>4</sup> As promised,  $\tau_t$  does not appear in the Ramsey planner's decision rule for  $\tau_{t+1}$ .

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  $\mu$ .

**Computing  $\mu$**  Define the selector vectors  $e_{\tau} = [0 \ 0 \ 1 \ 0]'$  and  $e_Q = [0 \ 1 \ 0 \ 0]'$  and express  $\tau_t = e_{\tau}' y_t$  and  $Q_t = e_Q' y_t$

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  $\Omega$  that satisfies

$$\Omega = \beta A_F' S A_F + \beta A_F' \Omega A_F \quad (3.183)$$

Equation (3.183) is a discrete Lyapunov equation that can be solved for  $\Omega$  using QuantEcon's `solve_discrete_lyapunov` function

The matrix  $F$  and therefore the matrix  $A_F = A - BF$  depend on  $\mu$

To find a  $\mu$  that guarantees that  $T_0 = G_0$  we proceed as follows:

1. Guess an initial  $\mu$ , compute a tentative Ramsey plan and the implied  $T_0 = y_0' \Omega(\mu) y_0$
2. If  $T_0 > G_0$ , lower  $\mu$ ; otherwise, raise  $\mu$
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 (3.168), (3.169), and (3.171)

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

$$w(Q_0, u_0 | \mu_0) = \sum_{t=0}^{\infty} \beta^t \left\{ A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 \right\} \quad (3.184)$$

where

- $\{Q_t, u_t\}_{t=0}^{\infty}$  are evaluated under the Ramsey plan whose recursive representation is given by (3.179), (3.180), (3.181)
- $\mu_0$  is the value of the Lagrange multiplier that assures budget balance, computed as described above

Evidently, these continuation values satisfy the recursion

$$w(Q_t, u_t | \mu_0) = A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 + \beta w(Q_{t+1}, u_{t+1} | \mu_0) \quad (3.185)$$

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 (3.179), (3.180), (3.181)

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

$$G_t = \beta^{-t} (G_0 - \sum_{s=1}^t \beta^s \tau_s Q_s) \quad (3.186)$$

where  $\{\tau_t, Q_t\}_{t=0}^{\infty}$  is the original Ramsey outcome<sup>5</sup>

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  $\mu$ , 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 (3.186)

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

$$w(Q_t, v(Q_t | \check{\mu}) | \check{\mu}) > w(Q_t, u_t | \mu_0) \quad (3.187)$$

Inequality (3.187) expresses a continuation Ramsey planner’s incentive to deviate from a time 0 Ramsey plan by

1. resetting  $u_t$  according to (3.178)
2. adjusting the Lagrange multiplier on the continuation appropriately to account for tax revenues already collected<sup>6</sup>

Inequality (3.187) expresses the time-inconsistency of a Ramsey plan

---

<sup>5</sup> 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  $s = 1, \dots, t$  under the original Ramsey plan.

<sup>6</sup> For example, let the Ramsey plan yield time 1 revenues  $Q_1 \tau_1$ . Then at time 1, a continuation Ramsey planner

**A Simulation** To bring out the time inconsistency of the Ramsey plan, we compare

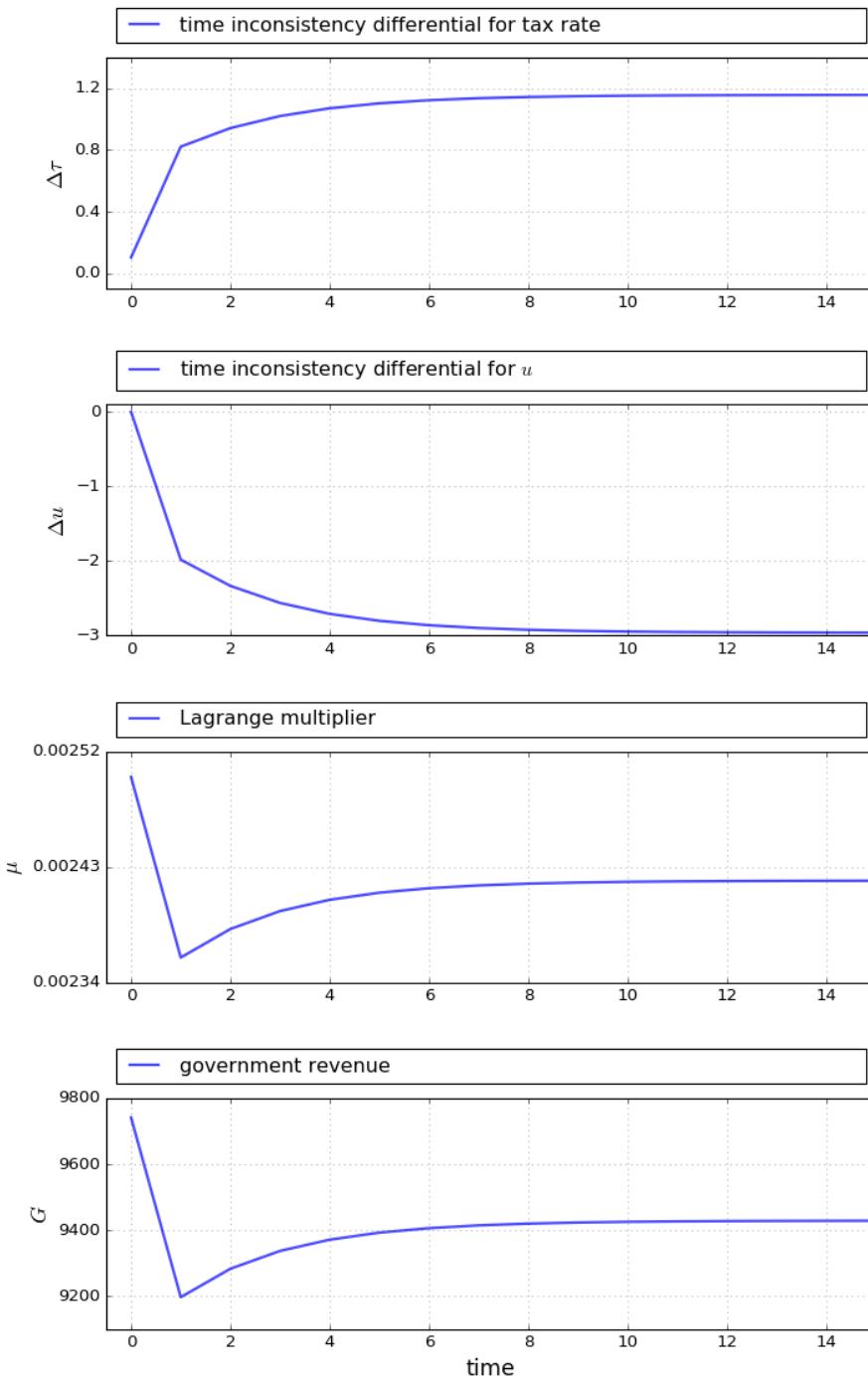
- the time  $t$  values of  $\tau_{t+1}$  under the original Ramsey plan with
- the value  $\check{\tau}_{t+1}$  associated with a new Ramsey plan begun at time  $t$  with initial conditions  $(Q_t, G_t)$  generated by following the *original* Ramsey plan

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

---

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  $\mu$  by applying the three-step procedure from the previous section to revenue requirements  $\tilde{G}_1$ .



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 (3.177):

$$\check{u}_t = -P_{22}^{-1}(\check{\mu}_t)P_{21}(\check{\mu}_t)z_t$$

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  $\mu_0$ <sup>7</sup>

To compute  $u_t$  along the Ramsey path, we just iterate the recursion starting (??) from the initial  $Q_0$  with  $u_0$  being given by formula (3.177)

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  $\tau_{t+1}$  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  $\tau$

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 be 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  $\mu$

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

---

<sup>7</sup> It can be verified that this formula puts non-zero weight only on the components 1 and  $Q_t$  of  $z_t$ .

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  $\tau_{t+1}$

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 (3.185), to define the continuation value function for the government:

$$J_t = A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 + \beta J_{t+1}(\tau_{t+1}, G_{t+1}) \quad (3.188)$$

This differs from (3.185) because

- continuation values are now allowed to depend explicitly on values of the choice  $\tau_{t+1}$ , 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  $\tau_t$  is chosen sequentially

Express the government budget constraint as requiring that  $G_0$  solves the difference equation

$$G_t = \beta \tau_{t+1} Q_{t+1} + \beta G_{t+1}, \quad t \geq 0 \quad (3.189)$$

subject to the terminal condition  $\lim_{t \rightarrow +\infty} \beta^t G_t = 0$

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  $(\tau_{t+1}, G_{t+1})$  subject to constraint (3.189)

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<sup>8</sup>

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

$$u_0 = v(Q_0, G_0, J_0), \quad (3.190)$$

choose subsequent taxes, outputs, and continuation values according to recursions that can be represented as

$$\hat{\tau}_{t+1} = \tau(Q_t, u_t, G_t, J_t) \quad (3.191)$$

$$u_{t+1} = \xi(Q_t, u_t, G_t, J_t, \tau_{t+1}) \quad (3.192)$$

$$G_{t+1} = \beta^{-1}G_t - \tau_{t+1}Q_{t+1} \quad (3.193)$$

$$J_{t+1}(\tau_{t+1}, G_{t+1}) = \nu(Q_t, u_t, G_{t+1}, J_t, \tau_{t+1}) \quad (3.194)$$

Here

- $\hat{\tau}_{t+1}$  is the time  $t + 1$  government action called for by the plan, while
- $\tau_{t+1}$  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

$$J_t = A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 + \beta J_{t+1}(\hat{\tau}_{t+1}, \hat{G}_{t+1}) \quad (3.195)$$

$$\geq A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 + \beta J_{t+1}(\tau_{t+1}, G_{t+1}) \quad (3.196)$$

for all tax rates  $\tau_{t+1} \in \mathbb{R}$  available to the government

$$\text{Here } \hat{G}_{t+1} = \frac{G_t - \hat{\tau}_{t+1} Q_{t+1}}{\beta}$$

- Inequality expresses that continuation values adjust to deviations in ways that discourage the government from deviating from the prescribed  $\hat{\tau}_{t+1}$
- Inequality (3.195) indicates that two continuation values  $J_{t+1}$  contribute to sustaining time  $t$  promised value  $J_t$

<sup>8</sup> This choice is the key to what [LS12] call ‘dynamic programming squared’.

- $J_{t+1}(\hat{\tau}_{t+1}, \hat{G}_{t+1})$  is the continuation value when the government chooses to confirm the private sector's expectation, formed according to the decision rule (3.191)<sup>9</sup>
- $J_{t+1}(\tau_{t+1}, G_{t+1})$  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 (3.195) 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 (3.195) 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 (3.191), (3.192), (3.193)

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 (3.191), (3.192), (3.193) 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

---

<sup>9</sup> Note the double role played by (3.191): as decision rule for the government and as the private sector's rule for forecasting government actions.

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 (3.195)

Recently, many have taken for granted that ‘optimal policy’ means ‘follow the Ramsey plan’<sup>10</sup>

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<sup>11</sup>
- 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<sup>12</sup>
- 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”<sup>13</sup>
- 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<sup>14</sup>

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

---

<sup>10</sup> 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.

<sup>11</sup> In our model, the Ramsey outcome would be a path  $(\vec{p}, \vec{Q})$ .

<sup>12</sup> See [GW10].

<sup>13</sup> Sometimes the analysis is framed in terms of following the Ramsey plan only from some future date  $T$  onwards.

<sup>14</sup> See [Bas05] and [ACK10].

## Contents

- *Optimal Taxation with State-Contingent Debt*
  - *Overview*
  - *A competitive equilibrium with distorting taxes*
  - *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
  - a flat rate tax on labor
  - trades in a full array of Arrow state contingent securities
- a representative consumer who values consumption and leisure
- a linear production function mapping leisure into a single good
- a Ramsey planner who at time  $t = 0$  chooses a plan for taxes and borrowing for all  $t \geq 0$

After first presenting the model in a space of sequences, we shall reformulate 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 the optimal responses of private agents to the Ramsey plan

See also [Optimal taxation](#) for an analysis within a linear-quadratic setting

## A competitive equilibrium with distorting taxes

For  $t \geq 0$ , the history  $s^t = [s_t, s_{t-1}, \dots, s_0]$  of an exogenous state  $s_t$  has joint probability density  $\pi_t(s^t)$ .

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  $s^t$

A representative household is endowed with one unit of time that can be divided between leisure  $\ell_t$  and labor  $n_t$ :

$$n_t(s^t) + \ell_t(s^t) = 1 \quad (3.197)$$

Output equals  $n_t(s^t)$  and can be divided between  $c_t(s^t)$  and  $g_t(s^t)$

$$c_t(s^t) + g_t(s^t) = n_t(s^t) \quad (3.198)$$

A representative household's preferences over  $\{c_t(s^t), \ell_t(s^t)\}_{t=0}^\infty$  are ordered by

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)] \quad (3.199)$$

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

$$g(s_t) = \tau_t(s^t) n_t(s^t) + \sum_{s_{t+1}} p_t(s_{t+1}|s^t) b_{t+1}(s_{t+1}|s^t) - b_t(s_t|s^{t-1}) \quad (3.200)$$

where

- $p_t(s_{t+1}|s^t)$  is the competitive equilibrium price of one-period Arrow state-contingent securities
- $b_t(s_t|s^{t-1})$  is government debt falling due at time  $t$ , history  $s^t$ .

Here  $p_t(s_{t+1}|s^t)$  is the price of one unit of consumption at date  $t+1$  in state  $s_{t+1}$  at date  $t$  and history  $s^t$

The initial government debt  $b_0(s_0)$  is given

The representative household has a sequence of budget constraints whose time  $t \geq 0$  component is

$$c_t(s^t) + \sum_{s_{t+1}} p_t(s_{t+1}|s^t) b_{t+1}(s_{t+1}|s^t) = [1 - \tau_t(s^t)] n_t(s^t) + b_t(s_t|s^{t-1}) \quad \forall t \geq 0. \quad (3.201)$$

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 (3.198) 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 (3.199) subject to (3.201) and (3.197) 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 (3.199)

**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  $\{q_t^0(s^t)\}_{t=0}^\infty$  to the Arrow securities prices  $\{p_t(s_{t+1}|s^t)\}_{t=0}^\infty$

$$q_{t+1}^0(s^{t+1}) = p_t(s_{t+1}|s^t)q_t^0(s^t)$$

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 (3.199) subject to the feasibility constraints (3.197) and (3.198) 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 (3.201) into another, we can obtain the household's present-value budget constraint:

$$\sum_{t=0}^{\infty} \sum_{s^t} q_t^0(s^t) c_t(s^t) = \sum_{t=0}^{\infty} \sum_{s^t} q_t^0(s^t) [1 - \tau_t(s^t)] w_t(s^t) n_t(s^t) + b_0 \quad (3.202)$$

Here  $\{q_t^0(s^t)\}_{t=1}^{\infty}$  can be interpreted as a time 0 Arrow-Debreu price system

The Arrow-Debreu price system is related to the system of Arrow securities prices through the recursion:

$$q_{t+1}^0(s^{t+1}) = p_t(s_{t+1}|s^t) q_t^0(s^t) \quad (3.203)$$

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

$$(1 - \tau_t(s^t)) = \frac{u_l(s^t)}{u_c(s^t)} \quad (3.204)$$

and

$$p_{t+1}(s_{t+1}|s^t) = \beta \pi(s_{t+1}|s^t) \left( \frac{u_c(s^{t+1})}{u_c(s^t)} \right) \quad (3.205)$$

where  $\pi(s_{t+1}|s^t)$  is the probability distribution of  $s_{t+1}$  conditional on history  $s^t$

Equation (3.205) implies that the Arrow-Debreu price system satisfies

$$q_t^0(s^t) = \beta^t \pi_t(s^t) \frac{u_c(s^t)}{u_c(s^0)} \quad (3.206)$$

Use the first-order conditions (3.204) and (3.205) to eliminate taxes and prices from (3.202) to derive the *implementability condition*

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) [u_c(s^t) c_t(s^t) - u_\ell(s^t) n_t(s^t)] - u_c(s^0) b_0 = 0. \quad (3.207)$$

The **Ramsey problem** is to choose a feasible allocation that maximizes

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), 1 - n_t(s^t)] \quad (3.208)$$

subject to (3.207)

**Solution details** First define a “pseudo utility function”

$$V[c_t(s^t), n_t(s^t), \Phi] = u[c_t(s^t), 1 - n_t(s^t)] + \Phi [u_c(s^t)c_t(s^t) - u_\ell(s^t)n_t(s^t)], \quad (3.209)$$

where  $\Phi$  is a Lagrange multiplier on the implementability condition (3.202).

Next form the Lagrangian

$$J = \sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) \left\{ V[c_t(s^t), n_t(s^t), \Phi] + \theta_t(s^t) [n_t(s^t) - c_t(s^t) - g_t(s^t)] \right\} - \Phi u_c(0)b_0 \quad (3.210)$$

where  $\{\theta_t(s^t); \forall s^t\}_{t \geq 0}$  is a sequence of Lagrange multipliers on the feasible conditions (3.198)

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  $\{\theta_t(s^t); \forall s^t\}_{t \geq 0}$ .

The first-order conditions for the Ramsey problem for periods  $t \geq 1$  and  $t = 0$ , respectively, are

$$\begin{aligned} c_t(s^t): (1 + \Phi)u_c(s^t) + \Phi [u_{cc}(s^t)c_t(s^t) - u_{\ell c}(s^t)n_t(s^t)] - \theta_t(s^t) &= 0, \quad t \geq 1, \\ n_t(s^t): -(1 + \Phi)u_\ell(s^t) - \Phi [u_{c\ell}(s^t)c_t(s^t) - u_{\ell\ell}(s^t)n_t(s^t)] + \theta_t(s^t) &= 0, \quad t \geq 1 \end{aligned} \quad (3.211)$$

and

$$\begin{aligned} c_0(s^0, b_0): (1 + \Phi)u_c(s^0, b_0) + \Phi [u_{cc}(s^0, b_0)c_0(s^0, b_0) - u_{\ell c}(s^0, b_0)n_0(s^0, b_0)] - \theta_0(s^0, b_0) \\ - \Phi u_{cc}(s^0, b_0)b_0 &= 0, \\ n_0(s^0, b_0): -(1 + \Phi)u_\ell(s^0, b_0) - \Phi [u_{c\ell}(s^0, b_0)c_0(s^0, b_0) - u_{\ell\ell}(s^0, b_0)n_0(s^0, b_0)] + \theta_0(s^0, b_0) \\ + \Phi u_{c\ell}(s^0, b_0)b_0 &= 0. \end{aligned} \quad (3.212)$$

It is instructive to use first-order conditions (3.211) for  $t \geq 1$  to eliminate the multiplier  $\theta_t(s^t)$

For convenience, we suppress the time subscript and the index  $s^t$  and obtain

$$\begin{aligned} (1 + \Phi)u_c(c, 1 - c - g) + \Phi [cu_{cc}(c, 1 - c - g) - (c + g)u_{\ell c}(c, 1 - c - g)] \\ = (1 + \Phi)u_\ell(c, 1 - c - g) + \Phi [cu_{c\ell}(c, 1 - c - g) - (c + g)u_{\ell\ell}(c, 1 - c - g)], \end{aligned} \quad (3.213)$$

where we have imposed conditions (3.197) and (3.198)

Equation (3.213) is one equation that can be solved to express the one unknown  $c$  as a function of the one exogenous variable  $g$

Notice that a counterpart to  $b_t(s_t | s^{t-1})$  does *not* appear in (3.213), so  $c$  does not depend on it

But things are different for time  $t = 0$

An analogous argument for the  $t = 0$  equations (3.212) 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 if we were to specify that government purchases are history dependent functions  $g_t(s^t)$  of the history of  $s^t$  rather than being a time-invariant function of  $s_t$ .

**Proposition:** If government purchases are equal after two histories  $s^t$  and  $\tilde{s}^j$  for  $t, j \geq 0$ , i.e., if

$$g_t(s_t) = g_j(\tilde{s}_j) = g,$$

then it follows from (3.213) that the optimal choices of consumption and leisure,  $(c_t(s^t), \ell_t(s^t))$  and  $(c_j(\tilde{s}^j), \ell_j(\tilde{s}^j))$ , 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 leading up to that outcome.

**The Ramsey allocation for a given  $\Phi$**  Temporarily take  $\Phi$  as given.

We shall compute  $c_0(s^0, b_0)$  and  $n_0(s^0, b_0)$  from the first-order conditions (3.212).

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 tell-tale sign of the *time-inconsistency* of a Ramsey plan.

$\Phi$  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  $\Phi$ .

**Further specialization** At this point, it is useful to specialize the model in the following way.

We assume that  $s$  is governed by a finite state Markov chain with states  $s \in [1, \dots, S]$  and transition matrix  $\Pi$ , where

$$\Pi(s'|s) = \text{Prob}(s_{t+1} = s' | s_t = s).$$

Also, assume that government purchases  $g$  are an exact time-invariant function  $g(s)$  of  $s$ .

We maintain these assumptions throughout the remainder of this lecture.

**Determining  $\Phi$**  We complete the Ramsey plan by computing the Lagrange multiplier  $\Phi$  on the implementability constraint (3.207)

Government budget balance restricts  $\Phi$  via the following line of reasoning.

The household's first-order conditions imply

$$(1 - \tau_t(s^t)) = \frac{u_l(s^t)}{u_c(s^t)} \tag{3.214}$$

and the implied one-period Arrow securities prices

$$p_{t+1}(s_{t+1}|s^t) = \beta \Pi(s_{t+1}|s_t) \frac{u_c(s^{t+1})}{u_c(s^t)}. \tag{3.215}$$

Substituting from (3.214), (3.215), and the feasibility condition (3.198) into the recursive version (3.201) of the household budget constraint gives

$$\begin{aligned} u_c(s^t)[n_t(s^t) - g_t(s^t)] + \beta \sum_{s_{t+1}} \Pi(s_{t+1}|s_t) u_c(s^{t+1}) b_{t+1}(s_{t+1}|s^t) \\ = u_l(s^t) n_t(s^t) + u_c(s^t) b_t(s_t|s^{t-1}). \end{aligned} \quad (3.216)$$

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  $\beta$  times the conditional expectation of  $x_{t+1}(s^{t+1})$  appears on the left side

Hence the equation shares much of the structure of a simple asset pricing equation with  $x_t$  being analogous to the price of the asset at time  $t$ .

We learned earlier that for a Ramsey allocation,  $c_t(s^t)$ ,  $n_t(s^t)$  and  $b_t(s_t|s^{t-1})$ , and therefore also  $x_t(s^t)$ , are each functions only of  $s_t$ , being independent of the history  $s^{t-1}$  for  $t \geq 1$ .

That means that we can express equation (3.216) as

$$u_c(s)[n(s) - g(s)] + \beta \sum_{s'} \Pi(s'|s) x'(s') = u_l(s)n(s) + x(s), \quad (3.217)$$

where  $s'$  denotes a next period value of  $s$  and  $x'(s')$  denotes a next period value of  $x$ .

Equation (3.217) is easy to solve for  $x(s)$  for  $s = 1, \dots, S$ .

If we let  $\vec{n}, \vec{g}, \vec{x}$  denote  $S \times 1$  vectors whose  $i$ th elements are the respective  $n, g$ , and  $x$  values when  $s = i$ , and let  $\Pi$  be the transition matrix for the Markov state  $s$ , then we can express as the matrix equation

$$\vec{u}_c(\vec{n} - \vec{g}) + \beta \Pi \vec{x} = \vec{u}_l \vec{n} + \vec{x}. \quad (3.218)$$

This is a system of  $S$  linear equations in the  $S \times 1$  vector  $x$ , whose solution is

$$\vec{x} = (I - \beta \Pi)^{-1} [\vec{u}_c(\vec{n} - \vec{g}) - \vec{u}_l \vec{n}]. \quad (3.219)$$

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

$$\vec{b} = \frac{\vec{x}}{\vec{u}_c}, \quad (3.220)$$

where division here means element-by-element division of the respective components of the  $S \times 1$  vectors  $\vec{x}$  and  $\vec{u}_c$ .

Here is a computational algorithm:

1. Start with a guess for the value for  $\Phi$ , then use the first-order conditions and the feasibility conditions to compute  $c(s_t), n(s_t)$  for  $s \in [1, \dots, S]$  and  $c_0(s_0, b_0)$  and  $n_0(s_0, b_0)$ , given  $\Phi$ . These are  $2(S+1)$  equations in  $2(S+1)$  unknowns.
2. Solve the  $S$  equations (3.219) for the  $S$  elements of  $\vec{x}$ . These depend on  $\Phi$ .
3. Find a  $\Phi$  that satisfies

$$u_{c,0}b_0 = u_{c,0}(n_0 - g_0) - u_{l,0}n_0 + \beta \sum_{s=1}^S \Pi(s|s_0)x(s) \quad (3.221)$$

by gradually raising  $\Phi$  if the left side exceeds the right side and lowering  $\Phi$  if the left side is smaller.

4. After computing a Ramsey allocation, we can recover the flat tax rate on labor from (3.204) and the implied one-period Arrow securities prices from (3.205)

In summary, when  $g_t$  is a time invariant function of a Markov state  $s_t$ , a Ramsey plan can be constructed by solving  $3S + 3$  equations in  $S$  components each of  $\vec{c}$ ,  $\vec{n}$ , and  $\vec{x}$  together with  $n_0, c_0$ , and  $\Phi$ .

**Time inconsistency** Let  $\{\tau_t(s^t)\}_{t=0}^\infty, \{b_{t+1}(s_{t+1}|s^t)\}_{t=0}^\infty$  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 planner is *not* a time  $t$ , history  $s^t$  Ramsey plan.

This means that a Ramsey plan is *not time consistent*

Another way to say the same thing is that the 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 (3.216) appears to be a purely “forward-looking” variable.

But  $x_t(s^t)$  is also a natural candidate for a state variable in a recursive formulation of the Ramsey problem

**Intertemporal delegation** To express a Ramsey plan recursively, we imagine that a time 0 Ramsey planner is followed by a sequence of continuation Ramsey planners at times  $t = 1, 2, \dots$

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 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 the 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

$$V(x, s) = \max_{n, \{x'(s')\}} u(n - g(s), 1 - n) + \beta \sum_{s' \in S} \Pi(s'|s) V(x', s') \quad (3.222)$$

where maximization over  $n$  and the  $S$  elements of  $x'(s')$  is subject to the single implementability constraint for  $t \geq 1$

$$x = u_c(n - g(s)) - u_l n + \beta \sum_{s' \in S} \Pi(s'|s) x'(s') \quad (3.223)$$

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 one  $x'(s')$  for each  $s' \in S$ .

Associated with a value function  $V(x, s)$  that solves Bellman equation are  $S + 1$  time-invariant policy functions

$$\begin{aligned} n_t &= f(x_t, s_t), \quad t \geq 1 \\ x_{t+1}(s_{t+1}) &= h(s_{t+1}; x_t, s_t), \quad s_{t+1} \in S, t \geq 1. \end{aligned} \quad (3.224)$$

**The Ramsey problem** The Bellman equation for the time 0 Ramsey planner is

$$W(b_0, s_0) = \max_{n_0, \{x'(s_1)\}} u(n_0 - g_0, 1 - n_0) + \beta \sum_{s_1 \in S} \Pi(s_1|s_0) V(x'(s_1), s_1) \quad (3.225)$$

where the maximization over  $n_0$  and the  $S$  elements of  $x'(s_1)$  is subject to the time 0 implementability constraint

$$u_{c,0}b_0 = u_{c,0}(n_0 - g_0) - u_{l,0}n_0 + \beta \sum_{s_1 \in \mathcal{S}} \Pi(s_1|s_0)x'(s_1) \quad (3.226)$$

coming from restriction (3.221)

Associated with a value function  $W(b_0, n_0)$  that solves Bellman equation are  $S + 1$  time 0 policy functions

$$\begin{aligned} n_0 &= f_0(b_0, s_0) \\ x_1(s_1) &= h_0(s_1; b_0, s_0). \end{aligned} \quad (3.227)$$

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 (3.224) for the time  $t \geq 1$  continuation Ramsey planners.

The value function  $V(x_t, s_t)$  of the time  $t$  continuation Ramsey planner equals  $E_t \sum_{\tau=t}^{\infty} \beta^{\tau-t} u(c_\tau, l_\tau)$ , where 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 (3.223) and a Lagrange multiplier  $\Phi_0$  to constraint (3.221).

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 (3.222) are

$$\beta \Pi(s'|s) V_x(x', s') - \beta \Pi(s'|s) \Phi_1 = 0 \quad (3.228)$$

for  $x'(s')$  and

$$(1 + \Phi_1)(u_c - u_l) + \Phi_1 [n(u_{ll} - u_{lc}) + (n - g(s))(u_{cc} - u_{lc})] = 0 \quad (3.229)$$

for  $n$ .

Given  $\Phi_1$ , equation (3.229) is one equation to be solved for  $n$  as a function of  $s$  (or of  $g(s)$ ).

Equation (3.228) implies  $V_x(x', s') = \Phi_1$ , while an envelope condition is  $V_x(x, s) = \Phi_1$ , so it follows that

$$V_x(x', s') = V_x(x, s) = \Phi_1(x, s). \quad (3.230)$$

For the time 0 problem on the right side of the Ramsey planner's Bellman equation (3.225), the first-order conditions are

$$V_x(x(s_1), s_1) = \Phi_0 \quad (3.231)$$

for  $x(s_1), s_1 \in \mathcal{S}$ , and

$$\begin{aligned} (1 + \Phi_0)(u_{c,0} - u_{n,0}) + \Phi_0 [n_0(u_{ll,0} - u_{lc,0}) + (n_0 - g(s_0))(u_{cc,0} - u_{cl,0})] \\ - \Phi_0(u_{cc,0} - u_{cl,0})b_0 = 0 \end{aligned} \quad (3.232)$$

Notice the similarities and differences between the first-order conditions for  $t \geq 1$  and for  $t = 0$ .

An additional term is present in (3.232) 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 (3.231) and (3.232) imply that  $\Phi_0 = \Phi_1$  and that

$$V_x(x_t, s_t) = \Phi_0 \quad (3.233)$$

for all  $t \geq 1$ .

When  $V$  is concave in  $x$ , this implies *state-variable degeneracy* along a Ramsey plan in the sense that for  $t \geq 1$ ,  $x_t$  will be a time-invariant function of  $s_t$ .

Given  $\Phi_0$ , this function mapping  $s_t$  into  $x_t$  can be expressed as a vector  $\vec{x}$  that solves equation 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 an earlier Ramsey planner or continuation Ramsey planner.

When government expenditures  $g_t$  are a time invariant function of a Markov state  $s_t$ , a Ramsey plan and associated Ramsey allocation feature marginal utilities of consumption  $u_c(s_t)$  that, given  $\Phi$ , for  $t \geq 1$  depend only on  $s_t$ , but that for  $t = 0$  depend on  $b_0$  as well.

This means that  $u_c(s_t)$  will be a time invariant function of  $s_t$  for  $t \geq 1$ , but except when  $b_0 = 0$ , a different function for  $t = 0$ .

This in turn means that prices of one period Arrow securities  $p_t(s_{t+1}|s_t) = p(s_{t+1}|s_t)$  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 workings of 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 uncertainty.

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  $g_3 = g_l = 0.2$ .

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

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

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 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  $\beta = 0.9$

---

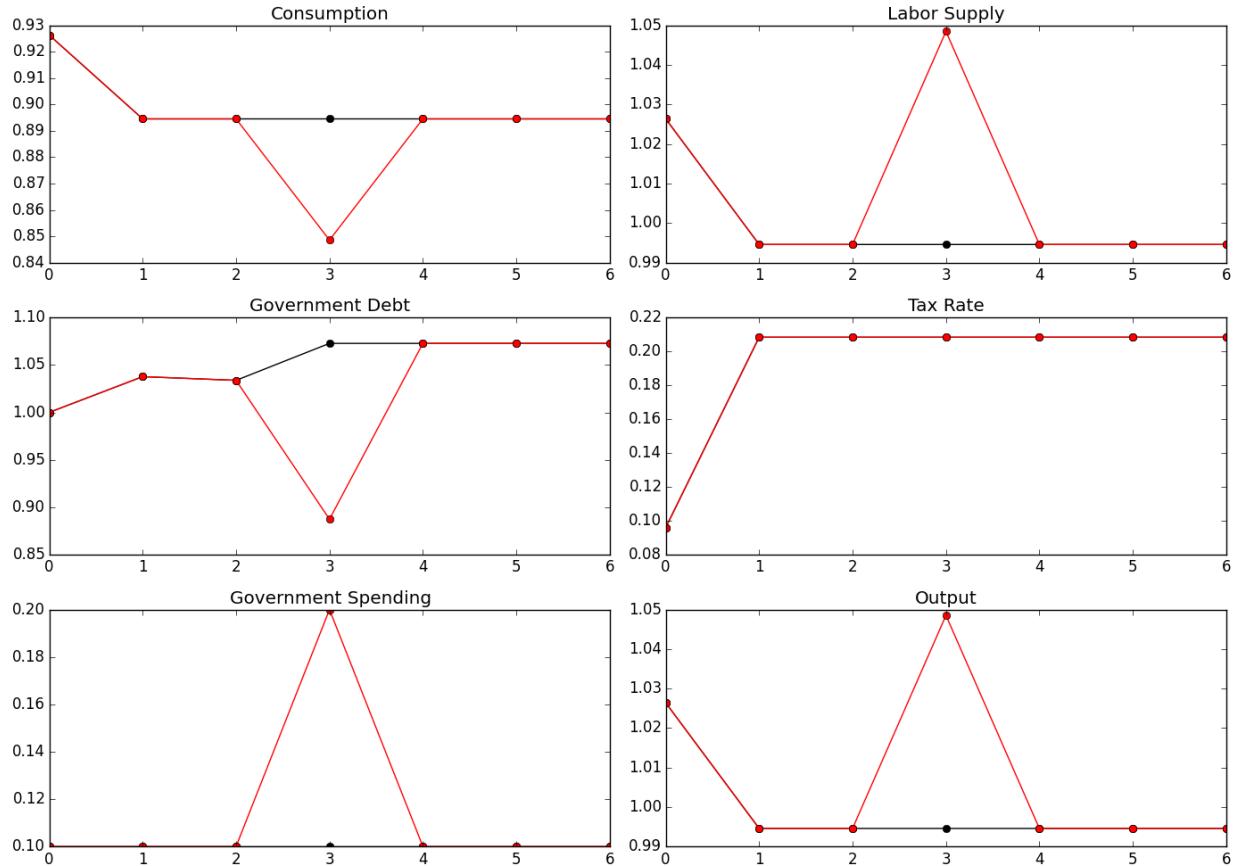
**Note:** For convenience in terms of matching our computer code, we have expressed utility as a function of  $n$  rather than leisure  $l$

---

We set initial government debt  $b_0 = 1$

The following figure plots the Ramsey tax under both realizations of the time  $t = 3$  government expenditure shock

- 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 gross interest rate for risk-free loans between periods  $t$  and  $t + 1$ , which equals

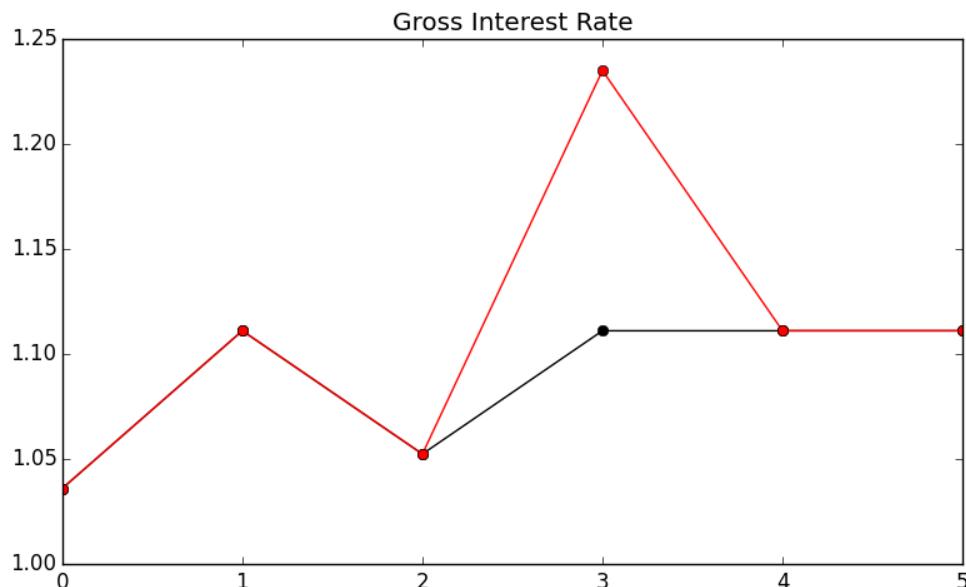
$$R_t = \frac{u_{c,t}}{\beta \mathbb{E}_t[u_{c,t+1}]}$$

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 then

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
- This time  $t = 4$  debt level is one that 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 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

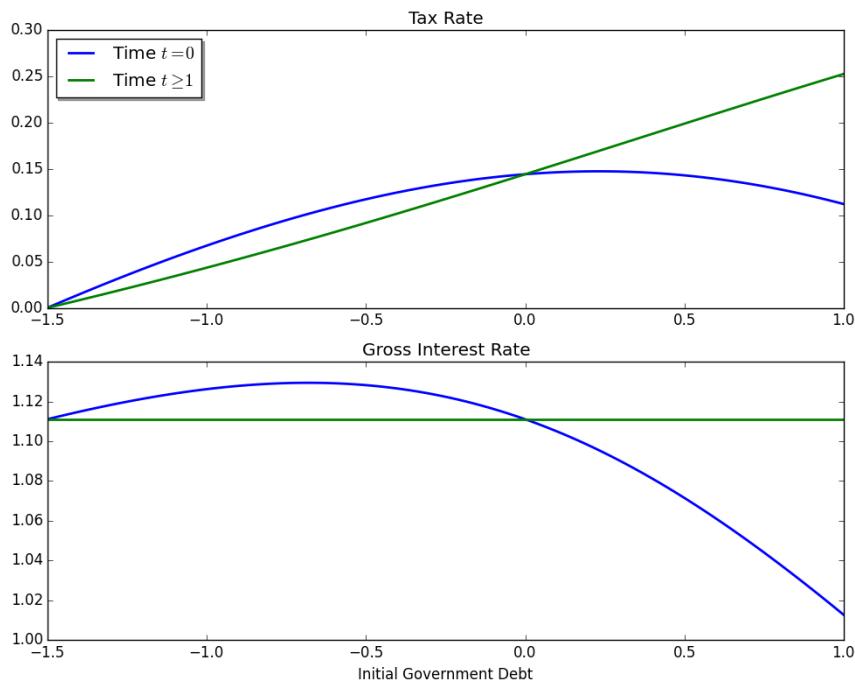
By doing this, it lowers the value of time  $t = 0$  debt that somehow it has inherited and must finance

**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  $t$

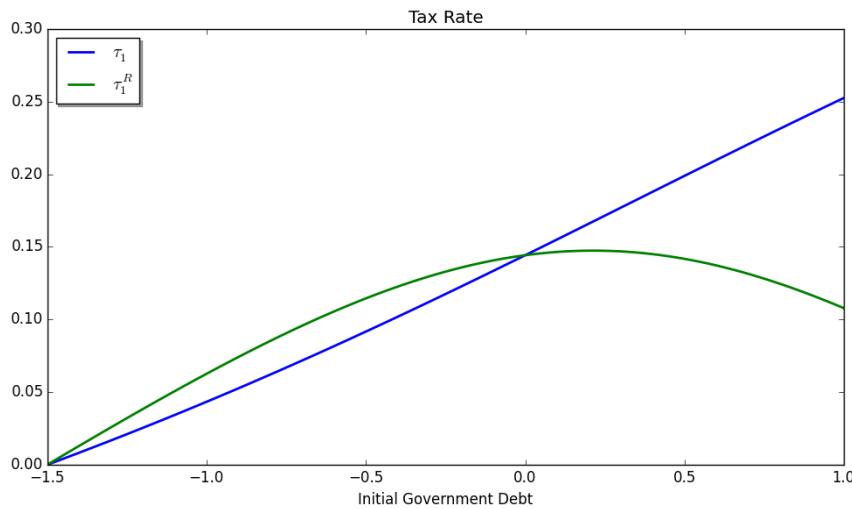
It is only for these two values of initial government debt that the Ramsey plan is time consistent.

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  $\tau_1^R$  the time  $t = 0$  tax rate chosen by the Ramsey planner confronting this value for initial government debt government

The figure below shows both the tax rate at time 1 chosen by our original Ramsey planner and what the new Ramsey planner would choose for its time  $t = 0$



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

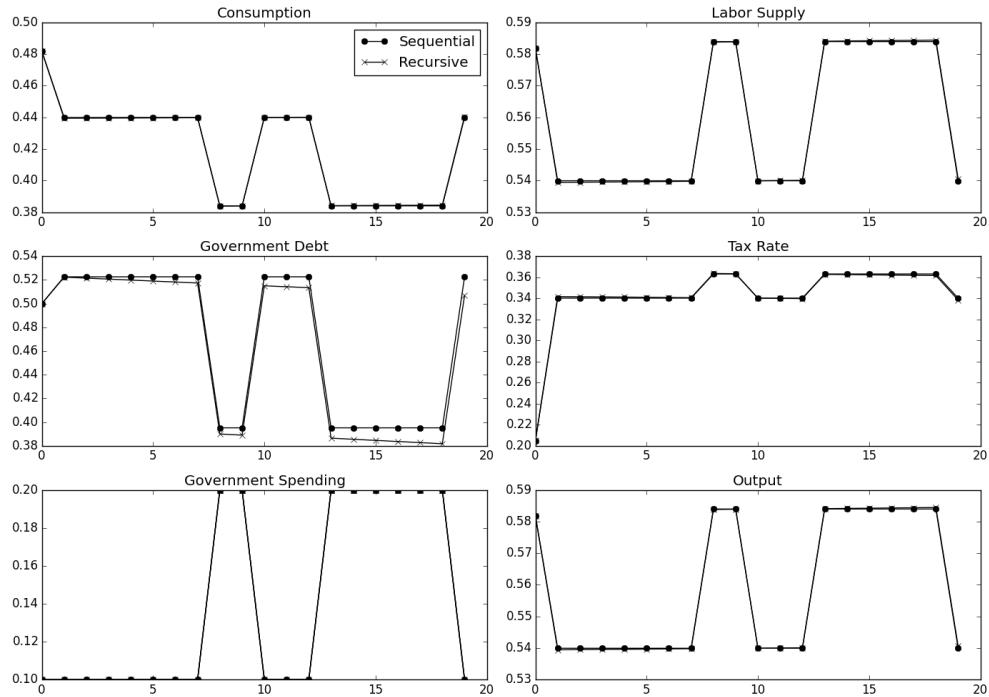
For other preferences, the tax rate may not be constant.

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` from the `QuantEcon` applications repo solves and simulates the Ramsey plan policy using both the recursive and sequential techniques.

We implement both techniques as classes that take as an input an object called `Para`

`Para` calibrates 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(s_0,T)

    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(s_0,T)

    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])

```

The file `main_LS.py` solves and plots the examples.

Calibrations are in `opt_tax_recur/calibrations/CES.py` and `calibrations/BGP.py`

**Further Comments** A related lecture describes an extension of the Lucas-Stokey model by Aiagari, Marcer, Sargent, and Seppälä (2002) [AMSS02]

In their (AMSS) economy, only a risk-free bond is traded

We compare the recursive representation of the Lucas-Stokey model presented in this lecture with the one in that lecture

By comparing these recursive formulations, we can glean a sense in which the dimension of the state is lower in the Lucas Stokey model.

Accompanying that difference in dimension are quite different dynamics of government debt

## 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 Lucas and Nancy Stokey [LS83].

Aiyagari, Marcet, Sargent, and Seppälä [AMSS02] (hereafter, AMSS) adapt this framework to study optimal taxation 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

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.

For  $t \geq 0$ , the history of the state is represented by  $s^t = [s_t, s_{t-1}, \dots, s_0]$ .

Government purchases  $g(s)$  are an exact time-invariant function of  $s$ .

Let  $c_t(s^t)$ ,  $\ell_t(s^t)$ , and  $n_t(s^t)$  denote consumption, leisure, and labor supply, respectively, at history  $s^t$  at time  $t$ .

A representative household is endowed with one unit of time that can be divided between leisure  $\ell_t$  and labor  $n_t$ :

$$n_t(s^t) + \ell_t(s^t) = 1. \quad (3.234)$$

Output equals  $n_t(s^t)$  and can be divided between consumption  $c_t(s^t)$  and  $g_t(s_t)$

$$c_t(s^t) + g_t(s_t) = n_t(s^t). \quad (3.235)$$

A representative household's preferences over  $\{c_t(s^t), \ell_t(s^t)\}_{t=0}^\infty$  are ordered by

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)], \quad (3.236)$$

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

**Risk-free debt only** In period  $t$  and history  $s^t$ , let

- $b_{t+1}(s^t)$  be the amount of government indebtedness carried over to and maturing in period  $t + 1$ , denominated in  $t + 1$  goods.
- $R_t(s^t)$  be the risk-free gross interest rate between periods  $t$  and  $t + 1$
- $T_t(s^t)$  be a nonnegative lump-sum transfer to the representative household <sup>1</sup>

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

$$\begin{aligned} b_t(s^{t-1}) &= \tau_t^n(s^t)n_t(s^t) - g_t(s_t) - T_t(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)} \\ &\equiv z(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)}, \end{aligned} \quad (3.237)$$

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 (3.237) yields:

$$b_t(s^{t-1}) = z(s^t) + \sum_{s^{t+1}|s^t} \beta \pi_{t+1}(s^{t+1}|s^t) \frac{u_c(s^{t+1})}{u_c(s^t)} b_{t+1}(s^t). \quad (3.238)$$

That  $b_{t+1}(s^t)$  is the same for all realizations of  $s_{t+1}$  captures its *risk-free* quality

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 (3.238) 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 (3.238), 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  $s_{t+1}$ .

<sup>1</sup> 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.

These substitutions give rise to the following expression:

$$b_t(s^{t-1}) = z(s^t) + \sum_{s^{t+1}|s^t} \beta \pi_{t+1}(s^{t+1}|s^t) \frac{u_c(s^{t+1})}{u_c(s^t)} \left[ z(s^{t+1}) + \frac{b_{t+2}(s^{t+1})}{R_{t+1}(s^{t+1})} \right].$$

After similar repeated substitutions for all future occurrences of government indebtedness, and by invoking the natural debt limit, we arrive at:

$$\begin{aligned} b_t(s^{t-1}) &= \sum_{j=0}^{\infty} \sum_{s^{t+j}|s^t} \beta^j \pi_{t+j}(s^{t+j}|s^t) \frac{u_c(s^{t+j})}{u_c(s^t)} z(s^{t+j}) \\ &= \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} z(s^{t+j}). \end{aligned} \quad (3.239)$$

Restriction (3.239) is a sequence of *implementability constraints* on a Ramsey allocation in an AMSS economy.

Expression (3.239) at time  $t = 0$  and initial state  $s^0$ , is an *implementability constraint* on a Ramsey allocation in a Lucas-Stokey economy too:

$$b_0(s^{-1}) = \mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} z(s^j). \quad (3.240)$$

**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 (3.240) of our AMSS economy is exactly the same as the one for the Lucas-Stokey economy

While (3.240) is the only implementability condition arising from budget constraints in the complete markets economy, many more implementability conditions must be satisfied in the AMSS economy because there is no state-contingent debt.

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 (3.238).

**Ramsey problem without state-contingent debt** The Ramsey planner chooses an allocation that solves

$$\max_{\{c_t(s^t), b_{t+1}(s^t)\}} \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u(c_t(s^t), 1 - c_t(s^t) - g_t(s_t))$$

where the maximization is subject to

$$\mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} z(s^j) \geq b_0(s^{-1}) \quad (3.241)$$

and

$$\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} z(s^{t+j}) = b_t(s^{t-1}) \quad \forall s^t \quad (3.242)$$

given  $b_0(s^{-1})$

Here we have

- substituted the resource constraint into the utility function, and also
- 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 now reads

$$z(s^t) = \left[ 1 - \frac{u_\ell(s^t)}{u_c(s^t)} \right] [c_t(s^t) + g_t(s_t)] - g_t(s_t) - T_t(s^t). : \text{label : AMSS44}_p y$$

**Lagrangian formulation** Let  $\gamma_0(s^0)$  be a nonnegative Lagrange multiplier on constraint (3.241).

As in the Lucas-Stokey economy, this multiplier is strictly positive if the government must resort to distortionary taxation; otherwise it equals zero.

The force 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 (3.242)

Depending on how the constraints bind, these multipliers might be positive or negative,

$$\begin{aligned} \gamma_t(s^t) &\geq (\leq) 0 \quad \text{if the constraint binds in this direction} \\ \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} z(s^{t+j}) &\geq (\leq) b_t(s^{t-1}). \end{aligned}$$

A negative multiplier  $\gamma_t(s^t) < 0$  means that if we could relax constraint , 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 <sup>2</sup>

These features flow from the fact that the government cannot use state-contingent debt and therefore cannot allocate its indebtedness most efficiently across future states.

Apply two transformations to the Lagrangian.

Multiply constraint by  $u_c(s^0)$  and the constraints by  $\beta^t u_c(s^t)$ .

---

<sup>2</sup> 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  $\gamma_t(\tilde{s}^t) > 0$ .

Then the Lagrangian for the Ramsey problem can be represented as

$$\begin{aligned}
 J &= \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \left\{ u(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)) \right. \\
 &\quad \left. + \gamma_t(s^t) \left[ \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j u_c(s^{t+j}) z(s^{t+j}) - u_c(s^t) b_t(s^{t-1}) \right] \right\} \\
 &= \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \left\{ u(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)) \right. \\
 &\quad \left. + \Psi_t(s^t) u_c(s^t) z(s^t) - \gamma_t(s^t) u_c(s^t) b_t(s^{t-1}) \right\}, 
 \end{aligned} \tag{3.243}$$

where

$$\Psi_t(s^t) = \Psi_{t-1}(s^{t-1}) + \gamma_t(s^t) \quad \text{and} \quad \Psi_{-1}(s^{-1}) = 0. \tag{3.244}$$

In (3.243), 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

$$\begin{aligned}
 u_c(s^t) - u_\ell(s^t) + \Psi_t(s^t) \{ [u_{cc}(s^t) - u_{c\ell}(s^t)] z(s^t) + u_c(s^t) z_c(s^t) \} \\
 - \gamma_t(s^t) [u_{cc}(s^t) - u_{c\ell}(s^t)] b_t(s^{t-1}) = 0 
 \end{aligned} \tag{3.245}$$

and with respect to  $b_t(s^t)$ ,

$$\mathbb{E}_t \left[ \gamma_{t+1}(s^{t+1}) u_c(s^{t+1}) \right] = 0 \tag{3.246}$$

If we substitute  $z(s^t)$  from equation (3.241) and its derivative  $z_c(s^t)$  into first-order condition (3.245), we will 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 (3.245) 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 (3.245) may change over time in response to realizations of the state, while the multiplier  $\Phi$  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 from the Lucas-Stokey economy
- while adding measurability constraints on functions of tails of the allocations at each time and history.

These functions represent the present values of government surpluses.

We now explore how these constraints alter the 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

$$b_t(s^{t-1}) + (1 - \tau_t(s^t))n_t(s^t) = c_t(s^t) + b_{t+1}(s^t)/R_t(s^t). \quad (3.247)$$

The household's first-order conditions are  $u_{c,t} = \beta R_t \mathbb{E}_t u_{c,t+1}$  and  $(1 - \tau_t)u_{c,t} = u_{l,t}$ .

Using these to eliminate  $R_t$  and  $\tau_t$  from the budget constraint (3.247) gives

$$b_t(s^{t-1}) + \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)}n_t(s^t) = c_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t)}{u_{c,t}(s^t)} \quad (3.248)$$

or

$$u_{c,t}(s^t)b_t(s^{t-1}) + u_{l,t}(s^t)n_t(s^t) = u_{c,t}(s^t)c_t(s^t) + \beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t) \quad (3.249)$$

Now define

$$x_t \equiv \beta b_{t+1}(s^t) \mathbb{E}_t u_{c,t+1} = u_{c,t}(s^t) \frac{b_{t+1}(s^t)}{R_t(s^t)} \quad (3.250)$$

and represent the household's budget constraint at time  $t$ , history  $s^t$  as

$$\frac{u_{c,t}x_{t-1}}{\beta \mathbb{E}_{t-1} u_{c,t}} = u_{c,t}c_t - u_{l,t}n_t + x_t \quad (3.251)$$

for  $t \geq 1$ .

**Measurability constraints** Write equation (3.249) as

$$b_t(s^{t-1}) = c_t(s^t) - \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)}n_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t)}{u_{c,t}}, \quad (3.252)$$

The right side of equation (3.252) 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 the right side of equation (3.252) has to equal  $b_t(s^{t-1})$

The means that is has to be *measurable* with respect to  $s^{t-1}$

These are the *measurability constraints* that the AMSS model adds to the single time 0 implementation constraint imposed in the Lucas and Stokey model.

**Two Bellman equations** Let

- $V(x_-, s_-)$  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

$$V(x_-, s_-) = \max_{\{n(s), x(s)\}} \sum_s \Pi(s|s_-) [u(n(s) - g(s), 1 - n(s)) + \beta V(x(s), s)] \quad (3.253)$$

subject to the following collection of implementability constraints, one for each  $s \in \mathcal{S}$ :

$$\frac{u_c(s)x_-}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_c(\tilde{s})} = u_c(s)(n(s) - g(s)) - u_l(s)n(s) + x(s). \quad (3.254)$$

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

$$W(b_0, s_0) = \max_{n_0, x_0} u(n_0 - g_0, 1 - n_0) + \beta V(x_0, s_0) \quad (3.255)$$

subject to

$$u_{c,0}b_0 = u_{c,0}(n_0 - g_0) - u_{l,0}n_0 + x_0. \quad (3.256)$$

**Martingale supercedes state-variable degeneracy** Let  $\mu(s|s_-)\Pi(s|s_-)$  be a Lagrange multiplier on constraint (3.254) 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

$$\beta V_x(x(s), s) = \mu(s|s_-). \quad (3.257)$$

Applying the envelope theorem to Bellman equation (3.253) gives

$$V_x(x_-, s_-) = \sum_s \Pi(s|s_-) \mu(s|s_-) \frac{u_c(s)}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_c(\tilde{s})}. \quad (3.258)$$

Equations (3.257) and (3.258) imply that

$$V_x(x_-, s_-) = \sum_s \left( \Pi(s|s_-) \frac{u_c(s)}{\sum_{\tilde{s}} \Pi(\tilde{s}|s_-) u_c(\tilde{s})} \right) V_x(x(s), s). \quad (3.259)$$

Equation (3.259) states that  $V_x(x, s)$  is a *risk-adjusted martingale*

This means 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 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  $\mu(s|s_-) = \beta V_x(x(s), s)$  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  $V_x(x, s)$  in the Lucas-Stokey model is the source of the 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 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  $x_- = x(s) = \check{x}$ :

$$\frac{u_c(s)\check{x}}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}) u_c(\tilde{s})} = u_c(s)(n(s) - g(s)) - u_l(s)n(s) + \check{x}, \quad s = 1, 2. \quad (3.260)$$

We guess and verify that the scaled Lagrange multiplier  $\mu(s) = \mu$  is a constant independent of  $s$ .

At a fixed  $x$ , because  $V_x(x, s)$  must be independent of  $s_-$ , equation 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  $\mu$ .

For the continuation Ramsey planner facing implementability constraints (3.260), the first-order conditions with respect to  $n(s)$  become

$$\begin{aligned} & u_c(s) - u_l(s)\mu \times \\ & \left\{ \frac{\check{x}}{\beta \sum_{\tilde{s}} \Pi(\tilde{s}) u_c(\tilde{s})} (u_{cc}(s) - u_{cl}(s)) - u_c(s) - n(s)(u_{cc}(s) - u_{cl}(s)) - u_{lc}(s)n(s) + u_l(s) \right\} \\ & = 0. \quad (3.261) \end{aligned}$$

Equations (3.261) are four equations in the four unknowns  $n(s), s = 1, 2, \check{x}$ , and  $\mu$ .

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 fluctuations 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 desired risk-sharing 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  $x_t \rightarrow \check{x}$  as  $t \rightarrow \infty$ .

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` from the applications repository [QuantEcon.applications](#)

**Anticipated One-Period War** In our lecture on [optimal taxation with state contingent debt](#) we studied in a simple setting how the government manages uncertainty.

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](#)

As before 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  $g_3 = g_l = 0.1$ .

The 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](#)

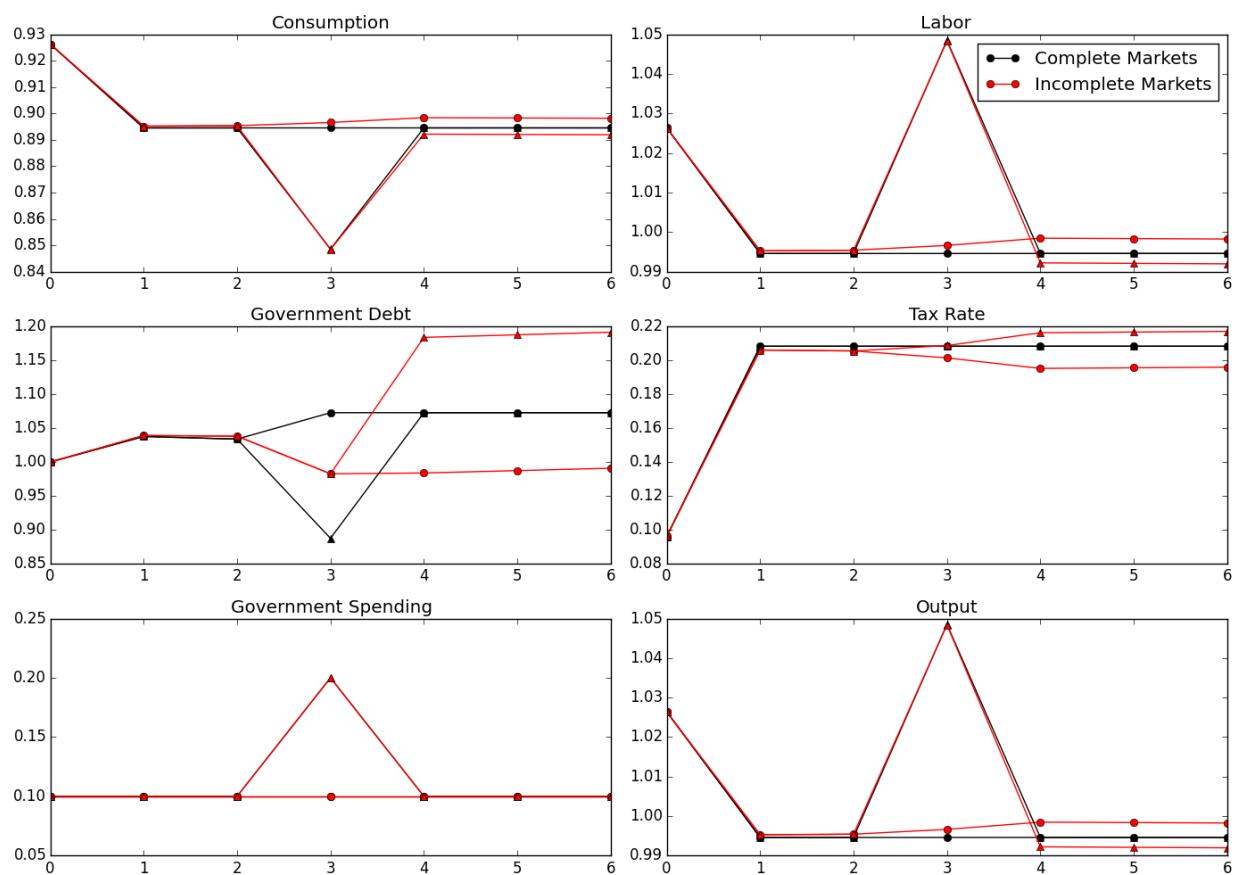
The following figure plots the Ramsey tax policy under both complete and incomplete markets for both possible realizations of the state at time  $t = 3$ .

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

How the Ramsey planner responds to war depending 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 facilitates 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 only of the current level of government spending, 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** The history dependence can be seen more starkly 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 BGP preferences

$$u(c, n) = \log(c) + 0.69 \log(1 - n)$$

so even with state contingent debt tax rates will not be constant.

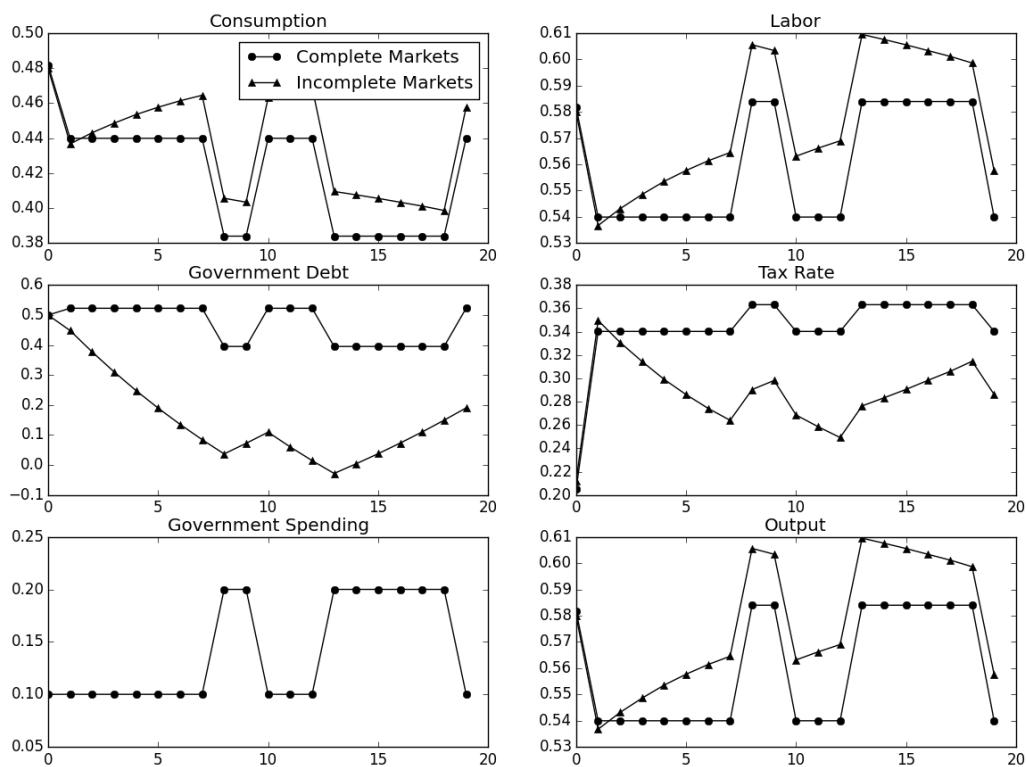
The figure below plots the optimal tax policies for both the case of state contingent (circles) and non state contingent debt (triangles).

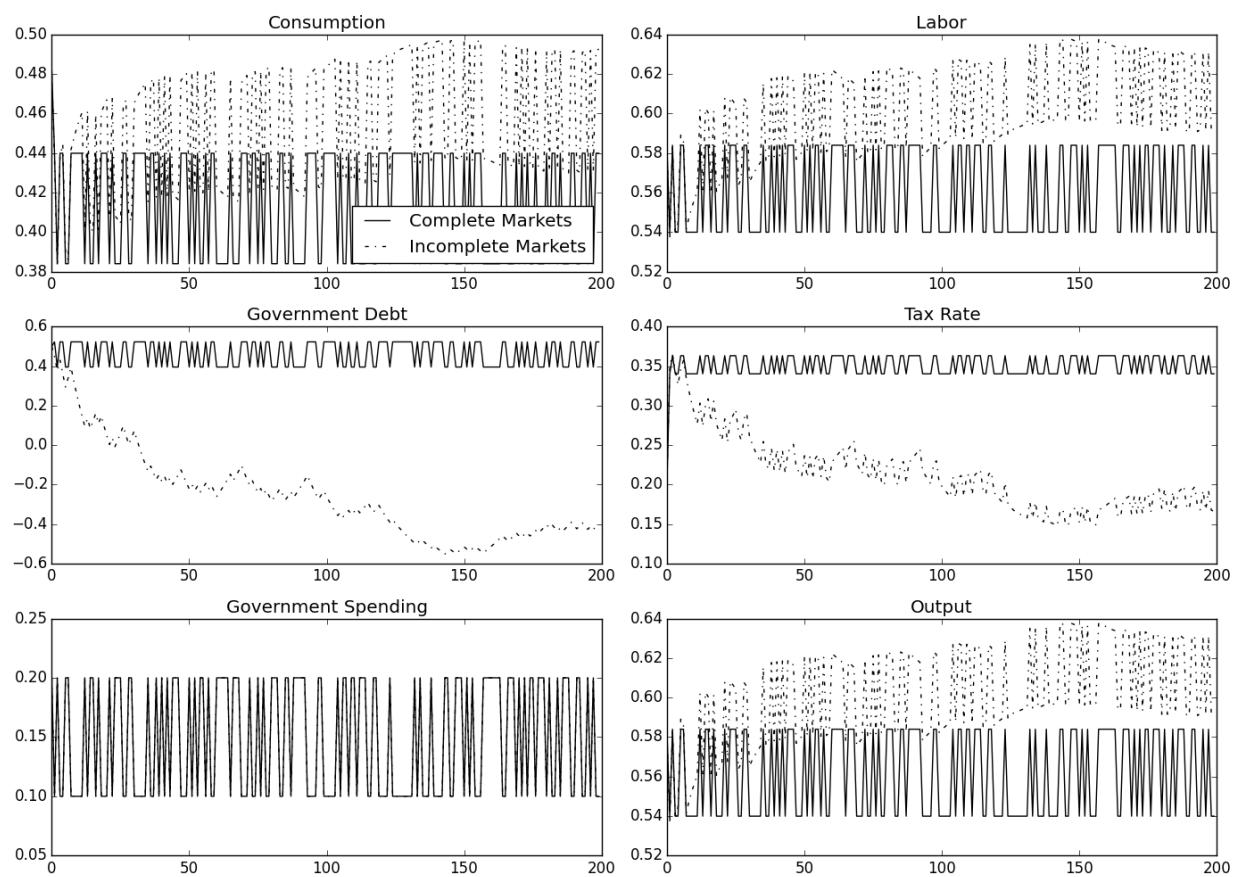
When the government experiences a prolonged period of peace, it is able to reduce government debt and permanently lower tax rates.

However, the government must finance a long war by borrowing and raising taxes.

This results in a drift away from the 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.





## 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \quad (3.262)$$

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 (3.262)

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
  - 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

$$c = y + B - q(B', y)B' \quad (3.263)$$

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  $y_t, y_{t-1}, \dots$ , at time  $t$
- can borrow or lend without limit in an international credit market at a constant international interest rate  $r$
- receive full payment if the government chooses to pay
- receive zero if the government defaults on its one-period debt due

When a government is expected to default next period with probability  $\delta$ , the expected value of a promise to pay one unit of consumption next period is  $1 - \delta$ .

Therefore, the discounted expected value of a promise to pay  $B$  next period is

$$q = \frac{1 - \delta}{1 + r} \quad (3.264)$$

### 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  $\theta$

### 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  $B$ ,
2. observes current output  $y$ , and
3. chooses either to
  - (a) default, or
  - (b) to honor  $B$  and set next period's assets to  $B'$

In a recursive formulation,

- state variables for the government comprise the pair  $(B, y)$
- $v(B, y)$  is the optimum value of the government's problem when at the beginning of a period it faces the choice of whether to honor or default
- $v_c(B, y)$  is the value of choosing to pay obligations falling due
- $v_d(y)$  is the value of choosing to default

$v_d(y)$  does not depend on  $B$  because, when access to credit is eventually regained, net foreign assets equal 0

Expressed recursively, the value of defaulting is

$$v_d(y) = u(h(y)) + \beta \int \{ \theta v(0, y') + (1 - \theta)v_d(y') \} p(y, y') dy'$$

The value of paying is

$$v_c(B, y) = \max_{B' \geq -Z} \left\{ u(y - q(B', y)B' + B) + \beta \int v(B', y') p(y, y') dy' \right\}$$

The three value functions are linked by

$$v(B, y) = \max\{v_c(B, y), v_d(y)\}$$

The government chooses to default when

$$v_c(B, y) < v_d(y)$$

and hence given  $B'$  the probability of default next period is

$$\delta(B', y) := \int \mathbb{1}\{v_c(B', y') < v_d(y')\} p(y, y') dy' \quad (3.265)$$

Given zero profits for foreign creditors in equilibrium, we can combine (3.264) and (3.265) to pin down the bond price function:

$$q(B', y) = \frac{1 - \delta(B', y)}{1 + r} \quad (3.266)$$

**Definition of equilibrium** An *equilibrium* is

- a pricing function  $q(B', y)$ ,
- a triple of value functions  $(v_c(B, y), v_d(y), v(B, y))$ ,
- a decision rule telling the government when to default and when to pay as a function of the state  $(B, y)$ , and
- an asset accumulation rule that, conditional on choosing not to default, maps  $(B, y)$  into  $B'$

such that

- The three Bellman equations for  $(v_c(B, y), v_d(y), v(B, y))$  are satisfied
- Given the price function  $q(B', y)$ , the default decision rule and the asset accumulation decision rule attain the optimal value function  $v(B, y)$ , and
- The price function  $q(B', y)$  satisfies equation (3.266)

## 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` from the `QuantEcon.applications` package 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
```

```
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]

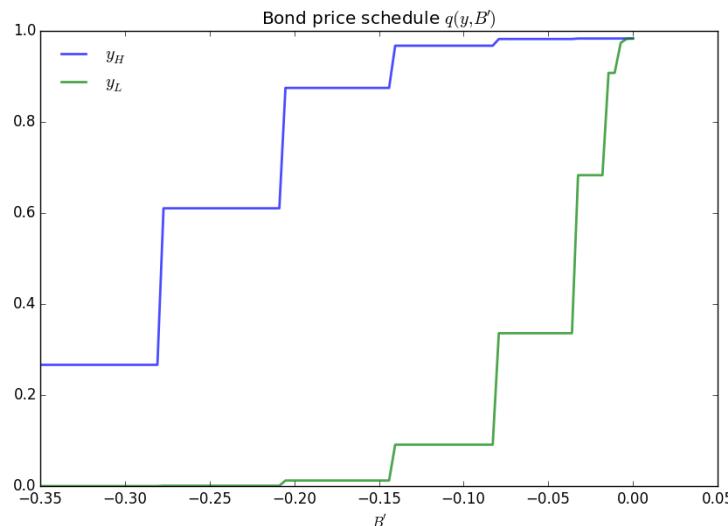
In what follows, all results are computed using Arellano's parameter values

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  $y$



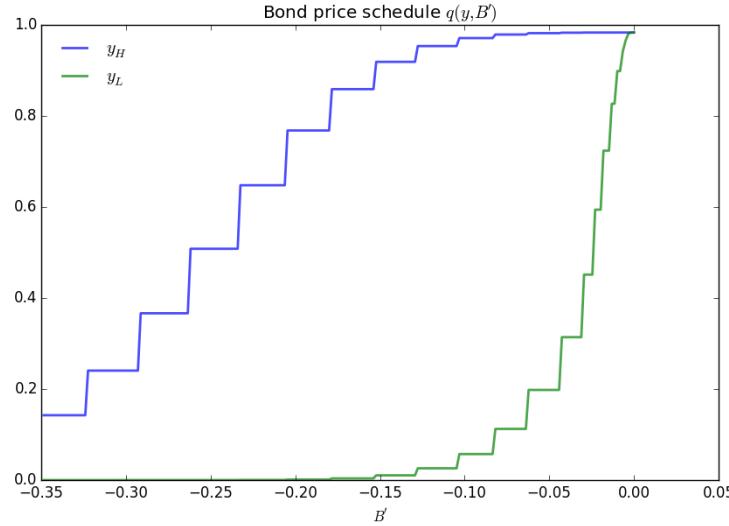
- $y_L$  is 5% below the mean of the  $y$  grid values
- $y_H$  is 5% above the mean of the  $y$  grid values

The grid used to compute this figure was relatively coarse ( $ny, nB = 21, 251$ ) in order to match Arrelano's findings

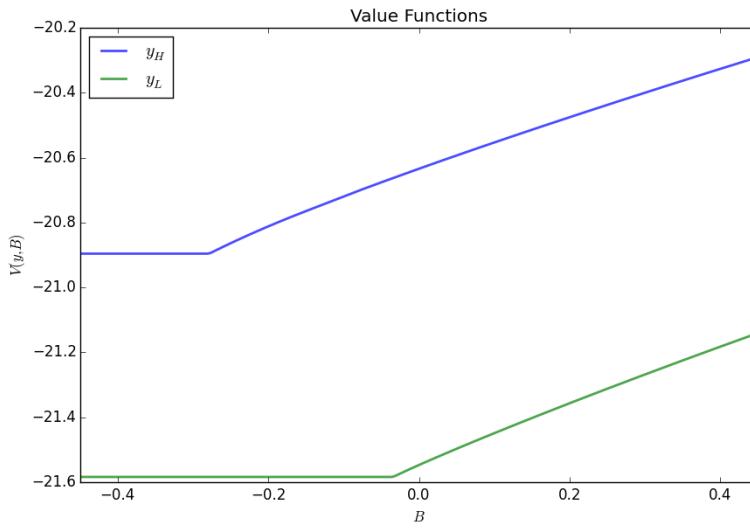
Here's the same relationships computed on a finer grid ( $ny, nB = 51, 551$ )

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 (3.265)

The next plot shows these default probabilities over  $(B', y)$  as a heat map

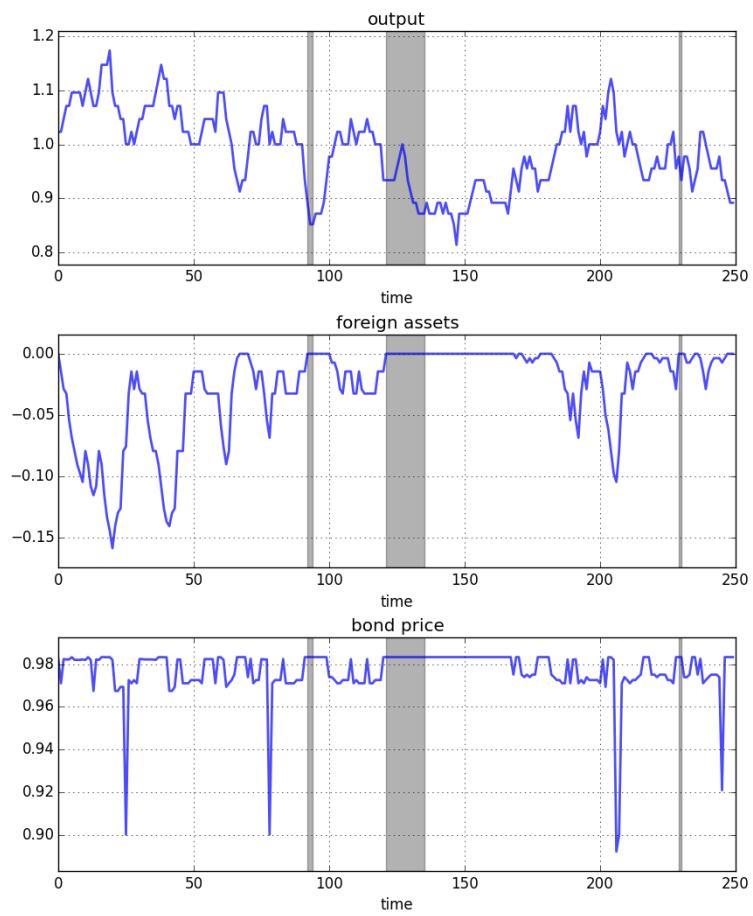
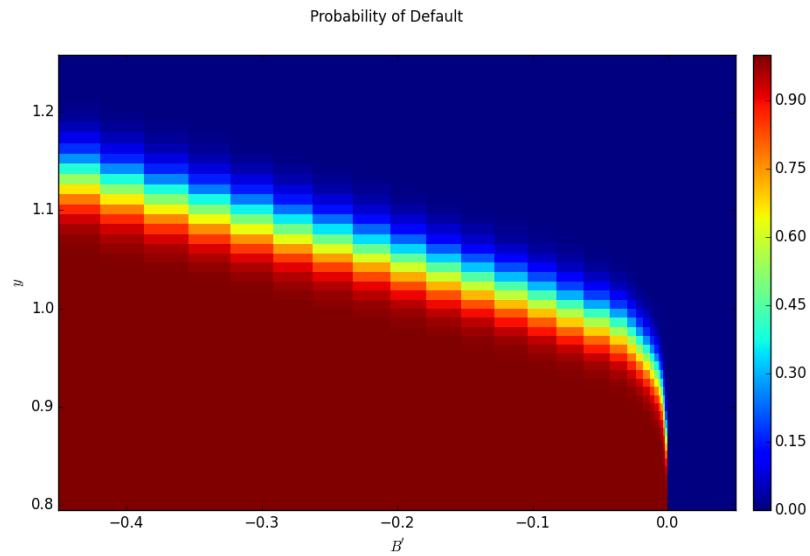
As anticipated, the probability that the government chooses to default in the following period increases with indebtedness and falls with income

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](#)



---

CHAPTER  
FOUR

---

## SOLUTIONS

Each lecture with exercises has a link to solutions immediately after the exercises

The links are to static versions of IPython Notebook files — the originals are located in matching topic folders contained in the '**QuantEcon.applications** <<https://github.com/QuantEcon/QuantEcon.applications>' repo. For example, to obtain a solutions notebook for the schelling chapter, it can be found [here](#)

If you look at a [typical solution notebook](#) you'll see a download icon on top right

You can download a copy of the ipynb file (the notebook file) using that icon

Now start IPython notebook and navigate to the downloaded ipynb file

Once you open it in IPython notebook it should be running live, allowing you to make changes



## FAQS / USEFUL RESOURCES

This page collects FAQs, useful links and commands

### FAQs

#### How do I install Python?

See [this lecture](#)

#### How do I start Python?

Run one of these commands in the system terminal (terminal, cmd, etc., depending on your OS)

- `python` — the basic Python shell (actually, **don't** use it, see the next command)
- `ipython` — a much better Python shell
- `ipython notebook` — start IPython Notebook on local machine

See [here](#) for more details on running Python

#### How can I get help on a Python command?

See [this discussion](#)

#### Where do I get all the Python programs from the lectures?

To import the quantecon library [this discussion](#)

To get all the code at once, visit our public applications code repository

- <https://github.com/QuantEcon/QuantEcon.applications>

See [this lecture](#) for more details on how to download the programs

## What's Git?

- See *this exercise*

## Other Resources

### IPython Magics

Common IPython commands (IPython magics)

- `run foo.py` — run file `foo.py`
- `pwd` — show present working directory
- `ls` — list contents of present working directory
- `cd dir_name` — change to directory `dir_name`
- `cd ..` — go back
- `load file_name.py` — load `file_name.py` into cell

### IPython Cell Magics

These are for use in the IPython notebook

- `%%file new_file.py` — put at top of cell to save contents as `new_file.py`
- `%%timeit` — time execution of the current cell

## Useful Links

- [Anaconda Python distribution](#)
- [Wakari](#) — cloud computing with IPython Notebook interface
- [Sagemath Cloud](#) — another cloud computing environment that runs Python
- [r/python](#) — Get your hit of Python news
- [Kevin Sheppard's Python course](#) — Includes an on line text and econometric applications
- [Python for Economists](#) — A course by Alex Bell
- [Quandl](#) — A Python interface to Quandl

**PDF Lectures**

A quick introduction to Python in slide format

- Lecture 1
- Lecture 2



## REFERENCES

- [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.
- [ACK10] Andrew Atkeson, Varadarajan V Chari, and Patrick J Kehoe. Sophisticated monetary policies\*. *The Quarterly journal of economics*, 125(1):47–89, 2010.
- [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 be-wley 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.
- [Bew77] Truman Bewley. The permanent income hypothesis: A theoretical formulation. *Journal of Economic Theory*, 16(2):252–292, 1977.
- [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.

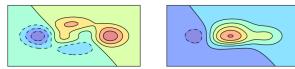
- [Cal78] Guillermo A. Calvo. On the time consistency of optimal policy in a monetary economy. *Econometrica*, 46(6):1411–1428, 1978.
- [Car01] Christopher D Carroll. A Theory of the Consumption Function, with and without Liquidity Constraints. *Journal of Economic Perspectives*, 15(3):23–45, 2001.
- [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.
- [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.

- [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.
- [HR87] Lars Peter Hansen and Scott F Richard. The Role of Conditioning Information in Deducing Testable. *Econometrica*, 55(3):587–613, May 1987.
- [HS00] Lars Peter Hansen and Thomas J Sargent. Wanting robustness in macroeconomics. *Manuscript, Department of Economics, Stanford University.*, 2000.
- [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 multi-period 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.
- [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.
- [Jud90] K L Judd. Cournot versus bertrand: A dynamic resolution. Technical Report, Hoover Institution, Stanford University, 1990.

- [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.
- [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.
- [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.
- [MP85] Rajnish Mehra and Edward C Prescott. The equity premium: A puzzle. *Journal of Monetary Economics*, 15(2):145–161, 1985.
- [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.
- [MZ75] Leonard J Mirman and Itzhak Zilcha. On optimal growth under uncertainty. *Journal of Economic Theory*, 11(3):329–339, 1975.
- [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.
- [Nea99] Derek Neal. The Complexity of Job Mobility among Young Men. *Journal of Labor Economics*, 17(2):237–261, 1999.
- [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.
- [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.
- [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.
- [Woo03] Michael Woodford. *Interest and Prices: Foundations of a Theory of Monetary Policy*. Princeton University Press, 2003.
- [YS05] G Alastair Young and Richard L Smith. *Essentials of statistical inference*. Cambridge University Press, 2005.



Acknowledgements: These lectures have benefitted greatly from comments and suggestion from our colleagues, students and friends. Special thanks go to Anmol Bhandari, Jeong-Hun Choi, Chase Coleman, David Evans, Chenghan Hou, Doc-Jin Jang, Spencer Lyon, Qingyin Ma, Matthew McKay, Tomohito Okabe, Alex Olssen, Nathan Palmer and Yixiao Zhou.

## INDEX

### A

A Simple Optimal Growth Model, 325  
An Introduction to Job Search, 234  
AR, 526  
ARMA, 523, 526  
ARMA Processes, 520

### B

Bellman Equation, 551  
Bisection, 142

### C

Central Limit Theorem, 248, 254  
Intuition, 254  
Multivariate Case, 258  
Classes of Assets, 408, 412  
cloud computing, 15, 181  
amazon ec2, 15  
google app engine, 15  
pythonanywhere, 15  
sagemath cloud, 15  
wakari, 15  
CLT, 248  
Complex Numbers, 524  
Continuous State Markov Chains, 443  
Covariance Stationary, 521  
Covariance Stationary Processes, 520

    AR, 522  
    MA, 522

Cython, 169, 177

### D

Data Sources, 155  
    World Bank, 158  
Dynamic Programming, 325, 327  
    Computation, 329  
    Shortest Paths, 230  
    Theory, 328

Unbounded Utility, 329

Value Function Iteration, 328, 329

Dynamic Typing, 170

### E

Eigenvalues, 183, 195  
Eigenvectors, 183, 195  
Ergodicity, 200, 214

### F

Finite Markov Asset Pricing  
    Lucas Tree, 413  
Finite Markov Chains, 200, 201  
    Stochastic Matrices, 201  
Fixed Point Theory, 463

### G

General Linear Processes, 522  
Git, 34

### H

History Dependent Public Policies, 607, 608  
    Competitive Equilibrium, 610  
    Ramsey Timing, 609  
    Sequence of Governments Timing, 610  
    Timing Protocols, 609

### I

Immutable, 95  
Integration, 138, 145  
IPython, 18, 161  
    Debugging, 164  
    Magics, 161  
    Reloading Modules, 163  
    Shell, 32  
    Timing Code, 162  
Irreducibility and Aperiodicity, 200, 207

- J
- Jupyter, 16, 18
  - Jupyter Notebook
    - Basics, 21
    - Figures, 24
    - Help, 25
    - nbviewer, 27
    - Setup, 19
    - Sharing, 27
- K
- Kalman Filter, 303
    - Programming Implementation, 309
    - Recursive Procedure, 307
- L
- Lake Model, 287
  - Law of Large Numbers, 248, 249
    - Illustration, 250
    - Multivariate Case, 258
    - Proof, 249
  - Linear Algebra, 138, 146, 183
    - Differentiating Linear and Quadratic Forms, 198
    - Eigenvalues, 195
    - Eigenvectors, 195
    - Matrices, 188
    - Matrix Norms, 198
    - Neumann's Theorem, 198
    - Positive Definite Matrices, 198
    - SciPy, 195
    - Series Expansions, 197
    - Spectral Radius, 198
    - Vectors, 184
  - Linear Markov Perfect Equilibria, 396
  - Linear State Space Models, 261
    - Distributions, 267, 268
    - Ergodicity, 272
    - Martingale Difference Shocks, 263
    - Moments, 267
    - Moving Average Representations, 267
    - Prediction, 277
    - Seasonals, 265
    - Stationarity, 272
    - Time Trends, 266
    - Univariate Autoregressive Processes, 264
    - Vector Autoregressions, 265
  - LLN, 248
- LQ Control, 346
  - Infinite Horizon, 356
  - Optimality (Finite Horizon), 349
- Lucas Model, 459
  - Assets, 459
  - Computation, 463
  - Consumers, 460
  - Dynamic Program, 461
  - Equilibrium Constraints, 461
  - Equilibrium Price Function, 462
  - Pricing, 460
  - Solving, 462
- M
- MA, 526
  - Marginal Distributions, 200, 205
  - Markov Asset Pricing, 407, 408
    - Overview, 408
  - Markov Chains, 201
    - Calculating Stationary Distributions, 212
    - Continuous State, 443
    - Convergence to Stationarity, 213
    - Cross-Sectional Distributions, 207
    - Ergodicity, 214
    - Forecasting Future Values, 214
    - Future Probabilities, 207
    - Irreducibility, Aperiodicity, 207
    - Marginal Distributions, 205
    - Simulation, 203
    - Stationary Distributions, 211
  - Markov Perfect Equilibrium, 394
    - Applications, 401
    - Background, 395
    - Overview, 394
  - Matplotlib, 10, 126
    - 3D Plots, 135
    - Multiple Plots on One Axis, 132
    - Object-Oriented API, 128
    - Simple API, 127
    - Subplots, 133
  - Matrix
    - Determinants, 193
    - Inverse, 193
    - Maps, 191
    - Numpy, 190
    - Operations, 189
    - Solving Systems of Equations, 191
  - McCall Model, 496

- Modeling  
 Career Choice, 477
- Models  
 Harrison Kreps, 420  
 Linear State Space, 263  
 Lucas Asset Pricing, 458  
 Markov Asset Pricing, 407  
 McCall, 496  
 On-the-Job Search, 486  
 Permanent Income, 428  
 Pricing, 408  
 Schelling's Segregation Model, 243
- Mutable, 95
- N**
- NetworkX, 14  
 Neumann's Theorem, 198  
 Newton-Raphson Method, 143  
 Nonparametric Estimation, 543  
 Numba, 169, 174  
 NumPy, 111, 112, 138  
 Arrays, 113  
 Arrays (Creating), 115  
 Arrays (Indexing), 116  
 Arrays (Methods), 118  
 Arrays (Operations), 119  
 Arrays (Shape and Dimension), 114  
 Comparisons, 123  
 Matrix Multiplication, 120  
 Universal Functions, 172  
 Vectorized Functions, 122
- O**
- Object Oriented Programming, 63  
 Classes, 66  
 Key Concepts, 64  
 Methods, 70  
 Special Methods, 78  
 Terminology, 64
- On-the-Job Search, 486, 487  
 Model, 487  
 Model Features, 487  
 Parameterization, 488  
 Programming Implementation, 489  
 Solving for Policies, 494
- Optimal Growth  
 Model, 326  
 Policy Function, 334
- Policy Function Approach, 327  
 Optimal Growth Part II: Adding Some Bling, 336
- Optimal Savings, 507  
 Computation, 509  
 Problem, 508  
 Programming Implementation, 510
- Optimal Taxation, 591  
 Optimization, 138, 145  
 Multivariate, 145
- Orthogonal Projection, 220
- P**
- Pandas, 13, 147  
 Accessing Data, 157  
 DataFrames, 150  
 Series, 148
- parallel computing, 15, 181  
 copperhead, 15  
 ipython, 15  
 pycuda, 15  
 starcluster, 15
- Periodograms, 538, 539  
 Computation, 540  
 Interpretation, 540
- Permanent Income Model, 428  
 Hall's Representation, 435  
 Savings Problem, 429
- Positive Definite Matrices, 198
- Pricing Models, 408  
 Finite Markov Asset Pricing, 412  
 Risk Aversion, 409  
 Risk Neutral, 408
- pyMC, 14  
 pyStan, 14  
 Python, 16  
 Anaconda, 17  
 Assertions, 97  
 common uses, 8  
 Comparison, 57  
 Conditions, 42  
 Content, 82  
 Cython, 177  
 Data Types, 49  
 Decorators, 100–102, 105  
 Descriptors, 100, 103  
 Dictionaries, 52  
 Docstrings, 60

- Exceptions, 98
- For loop, 39
- Functions, 41, 59
- Generator Functions, 106
- Generators, 105
- Handling Errors, 97
- Identity, 83
- import, 53
- Indentation, 40
- Interfacing with Fortran, 181
- Interpreter, 91
- Introductory Example, 35
- IO, 54
- IPython, 18
- Iterables, 86
- Iteration, 55, 85
- Iterators, 85, 86, 88
- keyword arguments, 61
- lambda functions, 61
- List comprehension, 44
- Lists, 38
- Logical Expressions, 58
- Matplotlib, 126
- Methods, 83
- Namespace (`__builtins__`), 93
- Namespace (Global), 92
- Namespace (Local), 92
- Namespace (Resolution), 93
- Namespaces, 89
- Nuitka, 182
- Numba, 174
- NumPy, 111
- Object Oriented Programming, 63
- Objects, 80
- Packages, 38
- Pandas, 147
- Parakeet, 182
- Paths, 55
- PEP8, 62
- Properties, 105
- PyPI, 16
- PyPy, 182
- Pyston, 182
- Pythran, 182
- Recursion, 110
- Runtime Errors, 98
- Scientific Libraries, 45
- SciPy, 125, 137
- Sets, 52
- Slicing, 52
- syntax and design, 9
- Tuples, 51
- Type, 81
- `urllib.request`, 156
- Variable Names, 88
- Vectorization, 171
- While loop, 41
- python, 7
- Q**
- QuantEcon, 27
  - Installation, 28
  - Repository, 28
- R**
- Ramsey Problem, 608, 612
  - Computing, 612
  - Credible Policy, 624
  - Optimal Taxation, 591
  - Recursive Representation, 615
  - Time Inconsistency, 620
  - Two Subproblems, 614
- Rational Expectations Equilibrium, 385, 386
  - Competitive Equilibrium (w. Adjustment Costs), 388
  - Computation, 391
  - Definition, 388
  - Planning Problem Approach, 391
- Robustness, 551
- S**
- Schelling Segregation Model, 243
- scientific programming, 9
  - Blaze, 16
  - CVXPY, 16
  - Jupyter, 16
  - Numba, 16
  - numeric, 10
  - PyTables, 16
- scikit-learn, 14
- SciPy, 125, 137, 138
  - Bisection, 142
  - Fixed Points, 145
  - Integration, 145
  - Linear Algebra, 146
  - Multivariate Root Finding, 145

Newton-Raphson Method, 143  
Optimization, 145  
Statistics, 139  
Smoothing, 538, 543  
Spectra, 538  
    Estimation, 538  
Spectra, Estimation  
    AR(1) Setting, 549  
    Fast Fourier Transform, 539  
    Pre-Filtering, 547  
    Smoothing, 543, 547  
Spectral Analysis, 520, 524  
Spectral Densities, 525  
Spectral Density, 526  
    interpretation, 526  
    Inverting the Transformation, 528  
    Mathematical Theory, 528  
Spectral Radius, 198  
Static Types, 170  
Stationary Distributions, 200, 211  
statsmodels, 14  
Stochastic Matrices, 201  
SymPy, 11

## T

Text Editors, 32

## U

Unbounded Utility, 329  
urllib.request, 156

## V

Value Function Iteration, 328  
Vectorization, 169, 171  
    Operations on Arrays, 172  
Vectors, 183, 184  
    Inner Product, 185  
    Linear Independence, 187  
    Norm, 185  
    Operations, 184  
    Span, 186

## W

White Noise, 521, 525  
Wold's Decomposition, 522