

# QUANTITATIVE ECONOMICS with Julia

Thomas Sargent and John Stachurski

October 09, 2015



## CONTENTS

<b>1 Programming in Julia</b>	<b>7</b>
1.1 Setting up Your Julia Environment . . . . .	7
1.2 An Introductory Example . . . . .	23
1.3 Julia Essentials . . . . .	33
1.4 Vectors, Arrays and Matrices . . . . .	46
1.5 Types, Methods and Performance . . . . .	59
1.6 Useful Libraries . . . . .	73
<b>2 Introductory Applications</b>	<b>87</b>
2.1 Linear Algebra . . . . .	87
2.2 Finite Markov Chains . . . . .	103
2.3 Shortest Paths . . . . .	123
2.4 Schelling's Segregation Model . . . . .	126
2.5 LLN and CLT . . . . .	131
2.6 Linear State Space Models . . . . .	144
2.7 A First Look at the Kalman Filter . . . . .	166
2.8 Uncertainty Traps . . . . .	178
2.9 Infinite Horizon Dynamic Programming . . . . .	185
2.10 LQ Dynamic Programming Problems . . . . .	201
2.11 Rational Expectations Equilibrium . . . . .	228
2.12 Markov Asset Pricing . . . . .	237
2.13 The Permanent Income Model . . . . .	247
<b>3 Advanced Applications</b>	<b>263</b>
3.1 Continuous State Markov Chains . . . . .	263
3.2 The Lucas Asset Pricing Model . . . . .	278
3.3 Modeling Career Choice . . . . .	287
3.4 On-the-Job Search . . . . .	296
3.5 Search with Offer Distribution Unknown . . . . .	307
3.6 Optimal Savings . . . . .	318
3.7 Robustness . . . . .	331
3.8 Covariance Stationary Processes . . . . .	356
3.9 Estimation of Spectra . . . . .	372
3.10 Optimal Taxation . . . . .	385
3.11 History Dependent Public Policies . . . . .	402

3.12 Default Risk and Income Fluctuations . . . . .	425
<b>References</b>	<b>441</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 orientated towards the website and to HTML output. As a result, the presentation quality can be less consistent than the website.



---

CHAPTER  
ONE

---

## PROGRAMMING IN JULIA

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

### 1.1 Setting up Your Julia Environment

#### Contents

- *Setting up Your Julia Environment*
  - *Overview*
  - *First Steps*
  - *IJulia*
  - *The QuantEcon Library*
  - *Exercises*

#### Overview

In this lecture we will cover how to get up and running with Julia

Topics:

1. Installation
2. Interactive Julia sessions
3. Running sample programs
4. Installation of libraries, including the Julia code that underpins these lectures

#### First Steps

**Installation** The first thing you will want to do is install Julia

The best option is probably to install the current release from the [download page](#)

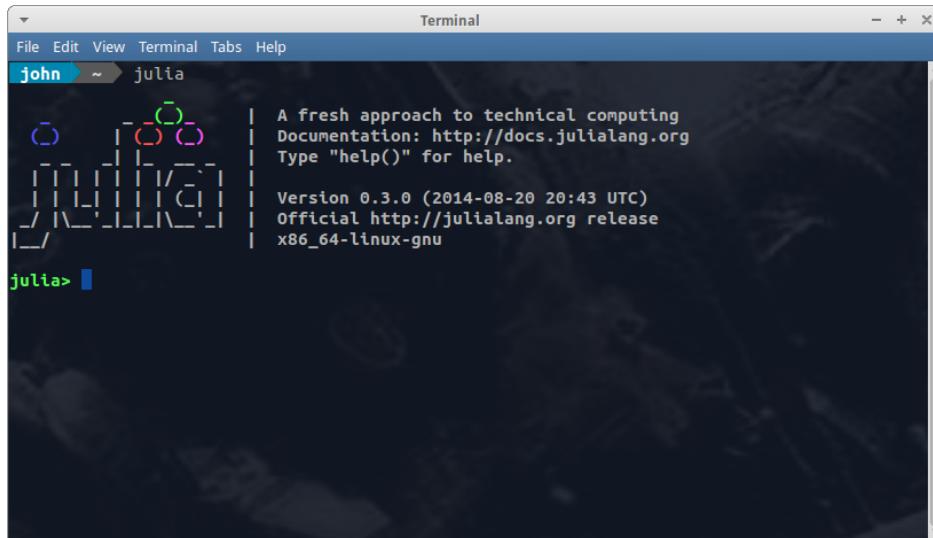
- Read through any download and installation instructions specific to your OS on that page

- Unless you have good reason to do otherwise, choose the current release rather than nightly build and the platform specific binary rather than source

Assuming there were no problems, you should now be able to start Julia either by

- navigating to Julia through your menus or desktop icons (Windows, OSX), or
- opening a terminal and typing `julia` (Linux)

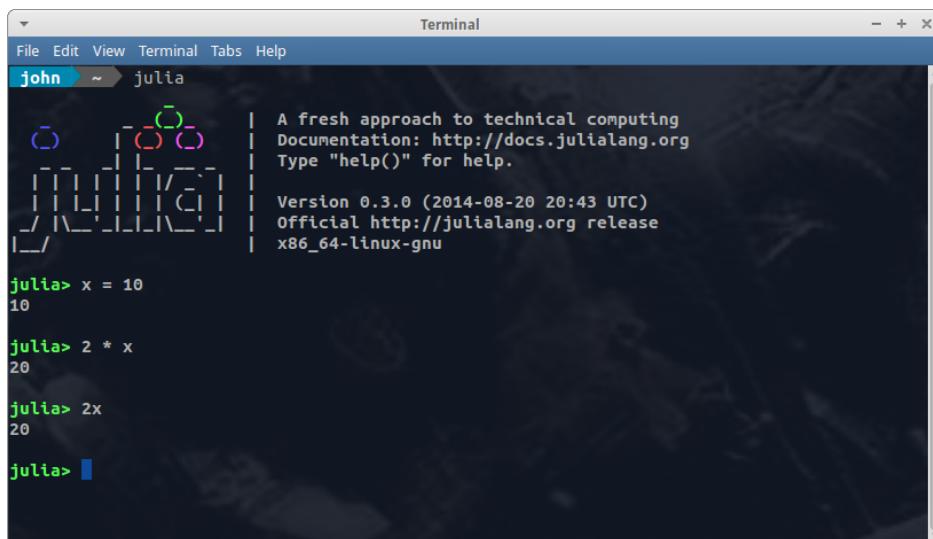
Either way you should now be looking at something like this (modulo your operating system — this is a Linux machine)



A screenshot of a terminal window titled "Terminal". The window has a blue header bar with the word "Terminal" and standard menu options: File, Edit, View, Terminal, Tabs, Help. Below the header is a dark-themed terminal window. In the top left corner of the terminal, it says "john ~ > julia". The terminal displays the following text:  
A fresh approach to technical computing  
Documentation: <http://docs.julialang.org>  
Type "help()" for help.  
Version 0.3.0 (2014-08-20 20:43 UTC)  
Official <http://julialang.org> release  
x86\_64-linux-gnu

The program that's running here is called the Julia REPL (Read Eval Print Loop) or Julia interpreter

Let's try some basic commands:



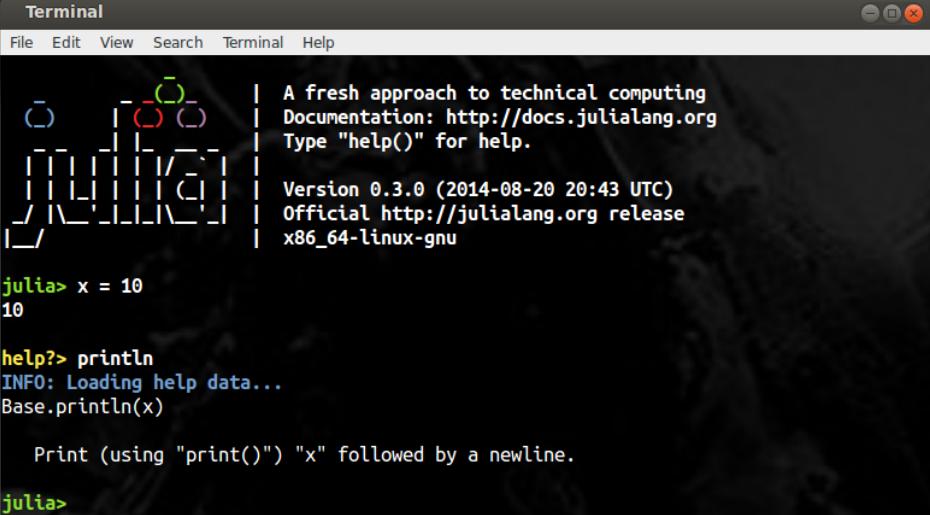
A screenshot of a terminal window titled "Terminal". The window has a blue header bar with the word "Terminal" and standard menu options: File, Edit, View, Terminal, Tabs, Help. Below the header is a dark-themed terminal window. In the top left corner of the terminal, it says "john ~ > julia". The terminal displays the following text:  
A fresh approach to technical computing  
Documentation: <http://docs.julialang.org>  
Type "help()" for help.  
Version 0.3.0 (2014-08-20 20:43 UTC)  
Official <http://julialang.org> release  
x86\_64-linux-gnu

At the bottom of the terminal, several commands are entered and their results are shown:  
`julia> x = 10`  
10  
`julia> 2 * x`  
20  
`julia> 2x`  
20  
`julia>`

The Julia interpreter has the kind of nice features you expect from a modern REPL

For example,

- Pushing the up arrow key retrieves the previously typed command
- If you type ? the prompt will change to help?> and give you access to online documentation



A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The main area displays the following text:

```

A fresh approach to technical computing
Documentation: http://docs.julialang.org
Type "help()" for help.

Version 0.3.0 (2014-08-20 20:43 UTC)
Official http://julialang.org release
x86_64-linux-gnu

julia> x = 10
10

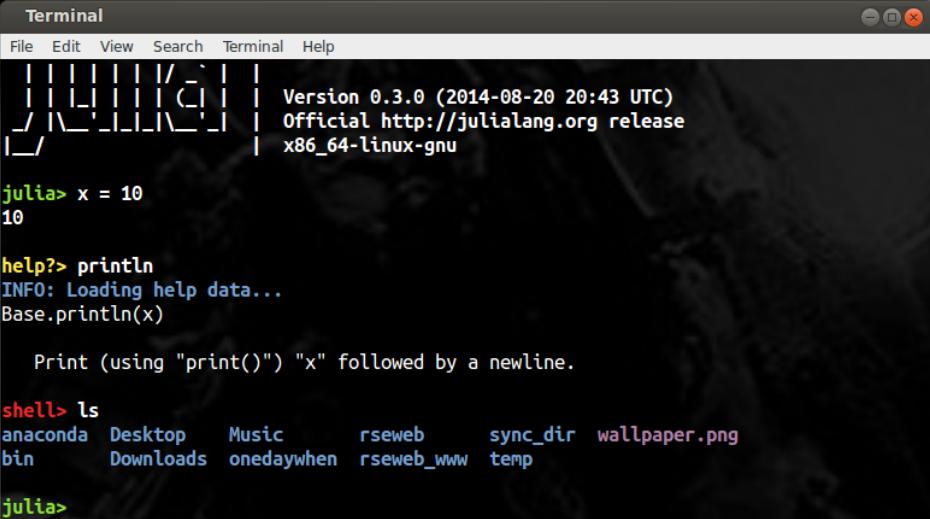
help?> println
INFO: Loading help data...
Base.println(x)

    Print (using "print()") "x" followed by a newline.

julia>

```

You can also type ; to get a shell prompt, at which you can enter shell commands



A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The main area displays the following text:

```

Version 0.3.0 (2014-08-20 20:43 UTC)
Official http://julialang.org release
x86_64-linux-gnu

julia> x = 10
10

help?> println
INFO: Loading help data...
Base.println(x)

    Print (using "print()") "x" followed by a newline.

shell> ls
anaconda Desktop Music rseweb sync_dir wallpaper.png
bin Downloads onedaywhen rseweb_www temp

julia>

```

(Here ls is a UNIX style command that lists directory contents — your shell commands depend on your operating system)

From now on instead of showing terminal images we'll show interactions with the interpreter as follows

```
julia> x = 10
10

julia> 2 * x
20
```

**Installing Packages** Julia includes many useful tools in the base installation

However, you'll quickly find that you also have need for at least some of the many external Julia code libraries

Fortunately these are very easy to install using Julia's excellent package management system

For example, let's install `DataFrames`, which provides useful functions and data types for manipulating data sets

```
julia> Pkg.add("DataFrames")
```

Assuming you have a working Internet connection this should install the `DataFrames` package

If you now type `Pkg.status()` you'll see `DataFrames` and its version number

To pull the functionality from `DataFrames` into the current session we type using `DataFrames`

```
julia> using DataFrames
```

Now let's use one of its functions to create a data frame object (something like an R data frame, or a spreadsheet)

```
julia> df = DataFrame(x1=[1, 2], x2=["foo", "bar"])
2x2 DataFrame
| Row | x1 | x2   |
|-----|----|-----|
| 1   | 1  | "foo" |
| 2   | 2  | "bar" |
```

One quick point before we move on: Running

```
julia> Pkg.update()
```

will update your installed packages and also update local information on the set of available packages

It's a good idea to make a habit of this

**Running Julia Scripts** Julia programs (or “scripts”) are text files containing Julia code, typically with the file extension `.jl`

Suppose we have a Julia script called `test_script.jl` that we wish to run

The contents of the file is as follows

```
for i in 1:3
    println("i = $i")
end
```

If that file exists in the present working directory we can run it with `include("test_script.jl")`

(To see what your present working directory is in a Julia session type `pwd()`)

Here's an example, where `test_script.jl` sits in directory `/home/john/temp`

```
julia> pwd()
"/home/john/temp"

julia> include("test_script.jl")
i = 1
i = 2
i = 3
```

(Of course paths to files will look different on different operating systems)

If the file is not in your `pwd` you can run it by giving the full path — in the present case

```
julia> include("/home/john/temp/test_script.jl")
```

Alternatively you can change your `pwd` to the location of the script

```
julia> cd("/home/john/temp")
```

and then run using `include("test_script.jl")` as before

**Editing Julia Scripts** Hopefully you can now run Julia scripts

You also need to know how to edit them

**Text Editors** Nothing beats the power and efficiency of a good text editor for working with program text

At a minimum, such an editor should provide

- syntax highlighting for the languages you want to work with
- automatic indentation
- text manipulation basics such as search and replace, copy and paste, etc.

There are many text editors that speak Julia, and a lot of them are free

Suggestions:

[Sublime Text](#) is a modern, popular and highly regarded text editor with a relatively moderate learning curve (not free but trial period is unlimited)

[Emacs](#) is a high quality free editor with a sharper learning curve

Finally, 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](#)

**IDEs** IDEs are Integrated Development Environments — they combine an interpreter and text editing facilities in the one application

For Julia one nice option is [Juno](#)

Alternatively there's IJulia, which is a little bit different again but has some great features that we now discuss

## IJulia

To work with Julia in a scientific context we need at a minimum

1. An environment for editing and running Julia code
2. The ability to generate figures and graphics

A very nice option that provides these features is [IJulia](#)

As a bonus, IJulia also provides

- Nicely formatted output in the browser, including tables, figures, animation, video, etc.
- The ability to mix in formatted text and mathematical expressions between cells
- Functions to generate PDF slides, static html, etc.

Whether you end up using IJulia as your primary work environment or not, you'll find learning about it an excellent investment

### Installing IJulia

IJulia is built on top of the [IPython notebook](#)

The IPython notebook started off as a Python tool but is in the process of being re-born as a language agnostic scientific programming environment (see [Jupyter](#))

The IPython notebook in turn has a range of dependencies that it needs to work properly

At present the easiest way to install all of these at once is to install the [Anaconda](#) Python distribution

**Installing Anaconda** Installing Anaconda is straightforward: [download the binary](#) and follow the instructions

If you are asked during the installation process whether you'd like to make Anaconda your default Python installation, say yes — you can always remove it later

Otherwise you can accept all of the defaults

Note that the packages in Anaconda update regularly — you can keep up to date by typing `conda update anaconda` in a terminal

### Installing IJulia

Just run

```
julia> Pkg.add("IJulia")
```

**Other Requirements** We'll be wanting to produce plots and while there are several options we'll start with PyPlot

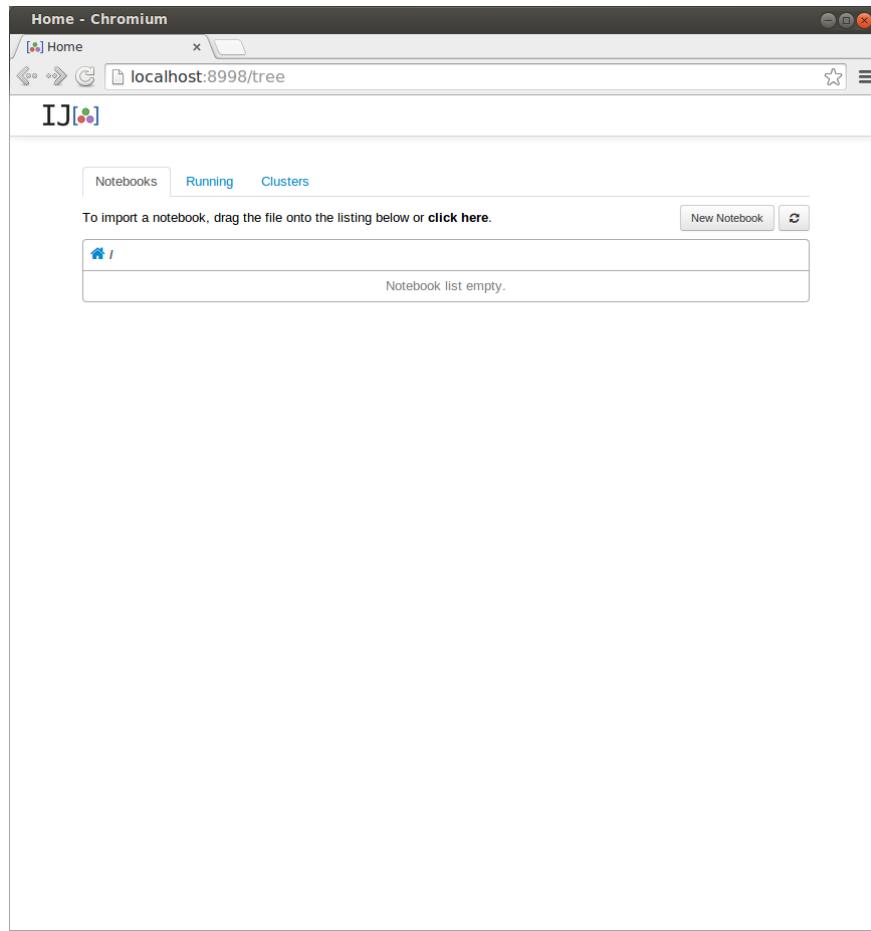
```
julia> Pkg.add("PyPlot")
```

Finally, since IJulia runs in the browser it might now be a good idea to update your browser  
One good option is to install a free modern browser such as [Chrome](#) or [Firefox](#)  
In our experience Chrome plays well with IJulia

**Getting Starting** To start IJulia in the browser, open up a terminal (or *cmd* in Windows) and type

```
ipython notebook --profile=julia
```

Here's an example of the kind of thing you should see



In this case the address is `localhost:8998/tree`, which indicates that the browser is communicating with a Julia session via port 8998 of the local machine

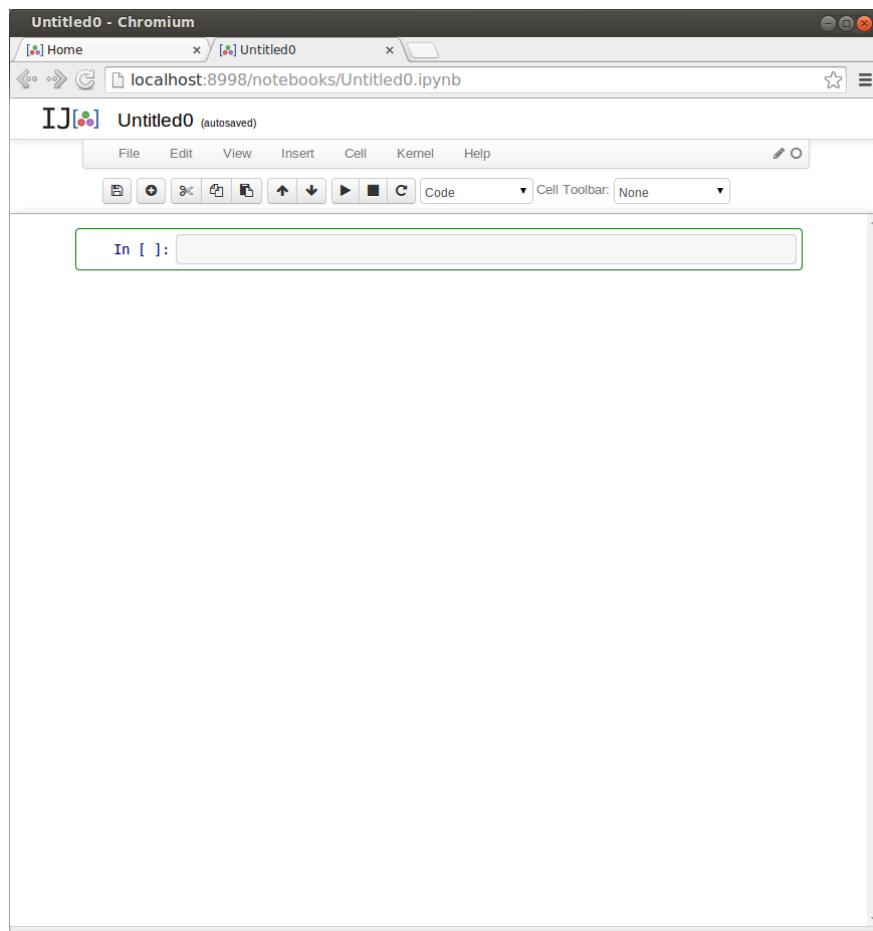
The page you are looking at is called the “dashboard”

From here you can now click on `New Notebook` and see something like this

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

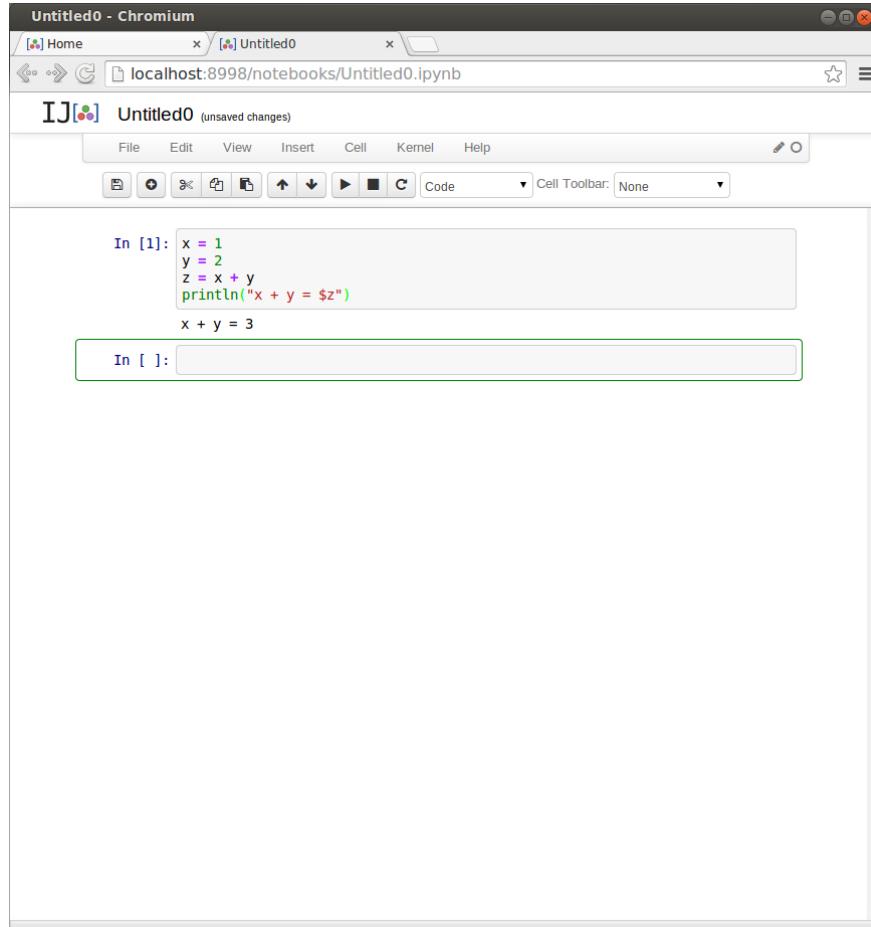
**Notebook Basics** Notice that in the previous figure the cell is surrounded by a green border

This means that the cell is in *edit mode*



As a result, you can type in Julia code and it will appear in the cell

When you're ready to execute these commands, hit Shift-Enter instead of the usual Enter



**Modal Editing** The next thing to understand about the IPython 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, as in the pictures above
  - 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

## Switching modes

- To switch to command mode from edit mode, hit the Esc key
- To switch to edit mode from command mode, hit Enter or click in a cell

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

For more details on the mechanics of using the notebook, see [here](#)

**Plots** As discussed above, IJulia integrates nicely with the plotting package `PyPlot.jl`

`PyPlot` in turn relies on the excellent Python graphics library `Matplotlib`

Once you have `PyPlot` installed you can load it via using `PyPlot`

We'll discuss plotting in detail later on but for now let's just make sure that it works

Here's a sample program you can run in IJulia

```
using PyPlot

n = 50
srand(1)
x = rand(n)
y = rand(n)
area = pi .* (15 .* rand(n)).^2 # 0 to 15 point radiiuses
scatter(x, y, s=area, alpha=0.5)
```

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 and Shift-Enter

This is what you should see

**Working with the Notebook** In this section we'll run you quickly through some more IPython notebook essentials — just enough so that we can press ahead with programming

**Tab Completion** A simple but useful feature of IJulia is tab completion

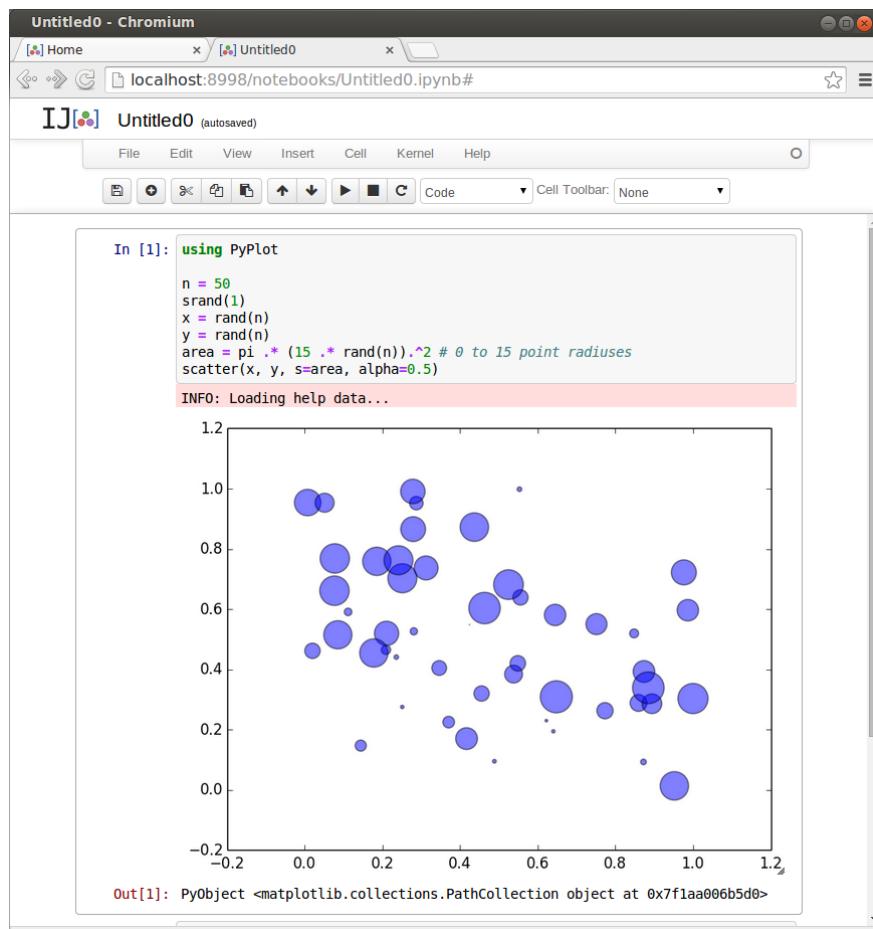
For example if you type `rep` and hit the tab key you'll get a list of all commands that start with `rep`

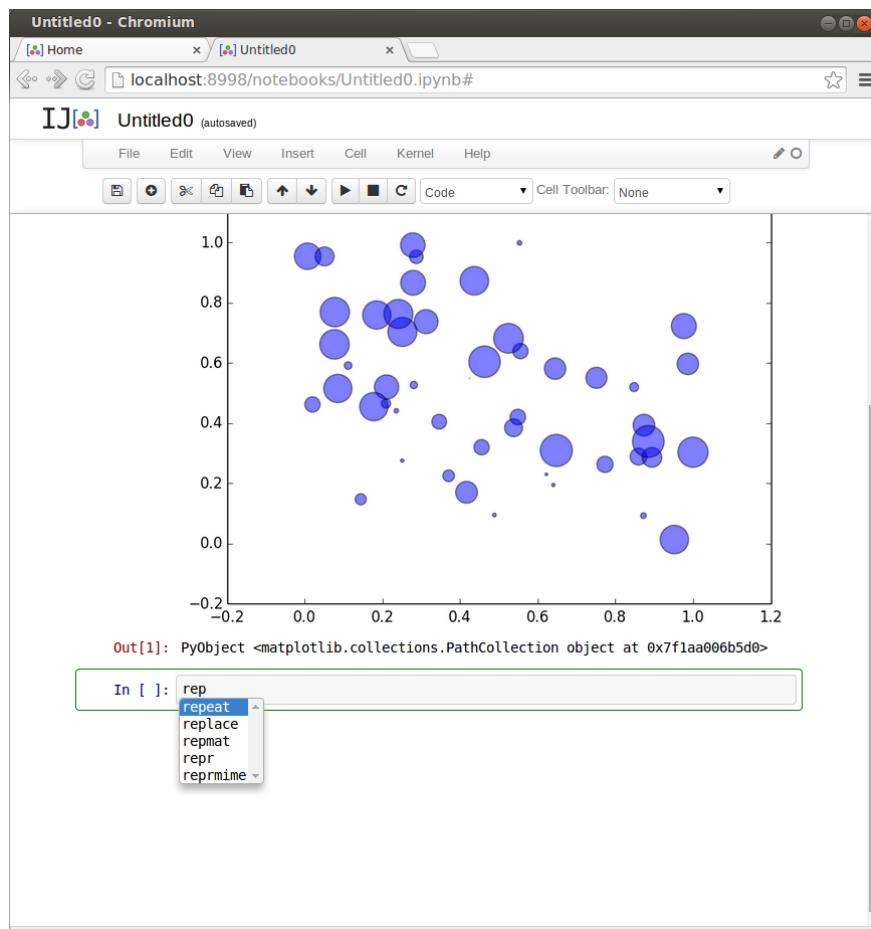
IJulia offers up the possible completions

This helps remind you of what's available and saves a bit of typing

**On-Line Help** To get help on the Julia function such as `repmat`, enter `help(repmat)`

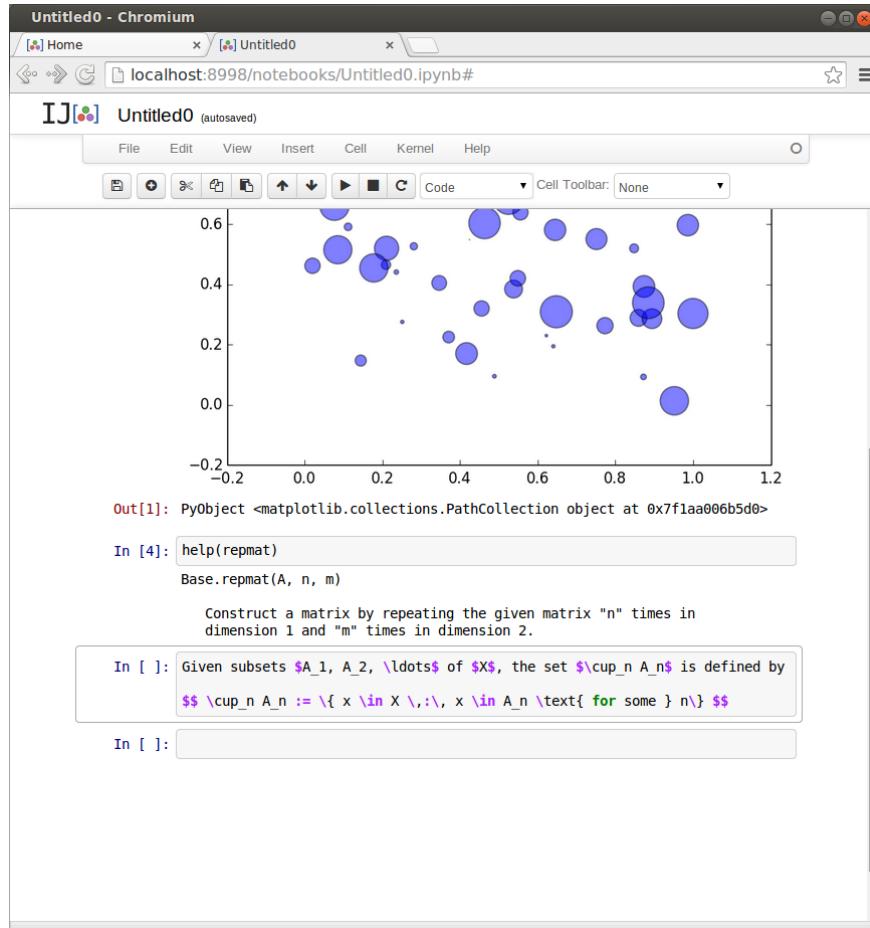
Documentation should now appear in the browser





**Other Content** In addition to executing code, the IPython 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)

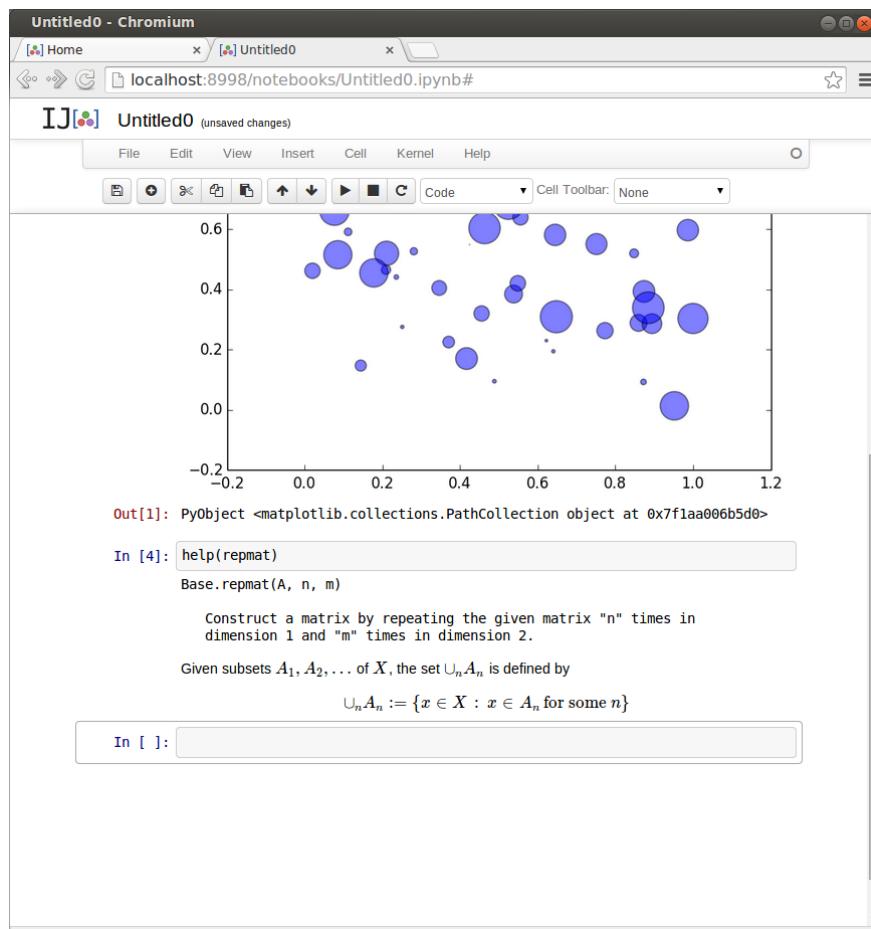
Now we Shift + Enter to produce this

**Shell Commands** You can execute shell commands (system commands) in IJulia by prepending a semicolon

For example, `;ls` will execute the UNIX style shell command `ls`, which — at least for UNIX style operating systems — lists the contents of the present working directory

These shell commands are handled by your default system shell and hence are platform specific

**Working with Files** To run an existing Julia file using the notebook we can either



1. copy and paste the contents into a cell in the notebook, or
2. use `include("filename")` in the same manner as for the Julia interpreter discussed above

More sophisticated methods for working with files are under active development and should be on-line soon

**Sharing Notebooks** Notebook files are just text files structured in [JSON](#) and typically ending with `.ipynb`

A notebook can easily be saved and shared between users — you just need to pass around the `ipynb` file

To open an existing `ipynb` file, import it from the dashboard (the first browser page that opens when you start IPython notebook) and run the cells or edit as discussed above

**nbviewer** The IPython organization has a site for sharing notebooks called [nbviewer](#)

The notebooks you see there are static HTML representations of notebooks

However, each notebook can be downloaded 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 we discussed just above

### The QuantEcon Library

The [QuantEcon library](#) is a community based code library containing open source code for quantitative economic modeling

Thanks to the heroic efforts of Spencer Lyon and some of his collaborators, this now includes a [Julia version](#)

You can install this package through the usual Julia package manager:

```
julia> Pkg.add("QuantEcon")
```

For example, the following code creates a discrete approximation to an AR(1) process

```
julia> using QuantEcon: tauchen
julia> tauchen(4, 0.9, 1)
([-6.88247, -2.29416, 2.29416, 6.88247],
4x4 Array{Float64,2}:
 0.945853  0.0541468  2.92863e-10  0.0
 0.00580845  0.974718   0.0194737  1.43534e-11
 1.43534e-11  0.0194737  0.974718   0.00580845
 2.08117e-27  2.92863e-10  0.0541468  0.945853  )
```

We'll learn much more about the library as we go along

**Installing via GitHub** You can also grab a copy of the files in the QuantEcon library directly by downloading the zip file — try clicking the “Download ZIP” button on the [main page](#)

Alternatively, you can get a copy of the repo using [Git](#)

For more information see [Exercise 1](#)

## Exercises

**Exercise 1** If you haven’t heard, [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

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 version
2. the point-and-click GUI versions
  - GUI style Git for Windows
  - GUI style Git for Mac

As an exercise, try getting a copy of the [QuantEcon repository](#) using Git

You can try the GUI options above or install the plain command line [Git](#)

If you’ve installed the command line version, open up a terminal and enter

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

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

Even better, sign up to [GitHub](#) — it’s free

Look into ‘forking’ GitHub repositories

(Loosely speaking, forking means making your own copy of a GitHub repository, stored on GitHub)

Try forking the [QuantEcon repository](#) for the course

Now try cloning it to some local directory, making edits, adding and committing them, and pushing 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](#)

## 1.2 An Introductory Example

### Contents

- *An Introductory Example*
  - *Overview*
  - *Example: Plotting a White Noise Process*
  - *Exercises*
  - *Solutions*

### Overview

We're now ready to start learning the Julia language itself

Our approach is aimed at those who already have at least some knowledge of programming — perhaps experience with Python, MATLAB, R, C or similar

In particular, we assume you have some familiarity with fundamental programming concepts such as

- variables
- loops
- conditionals (if/else)

If you have no such programming experience we humbly suggest you try Python first

Python is a great first language and, more importantly, there are many, many introductory treatments

In fact our treatment of Python is much slower than our treatment of Julia, especially at the start

Once you are comfortable with Python you'll find the leap to Julia is easy

**Approach** In this lecture we will write and then pick apart small Julia programs

At this stage the objective is to introduce you to basic syntax and data structures

Deeper concepts—how things work—will be covered in later lectures

Since we are looking for simplicity the examples are a little contrived

**Other References** The definitive reference is Julia's own documentation

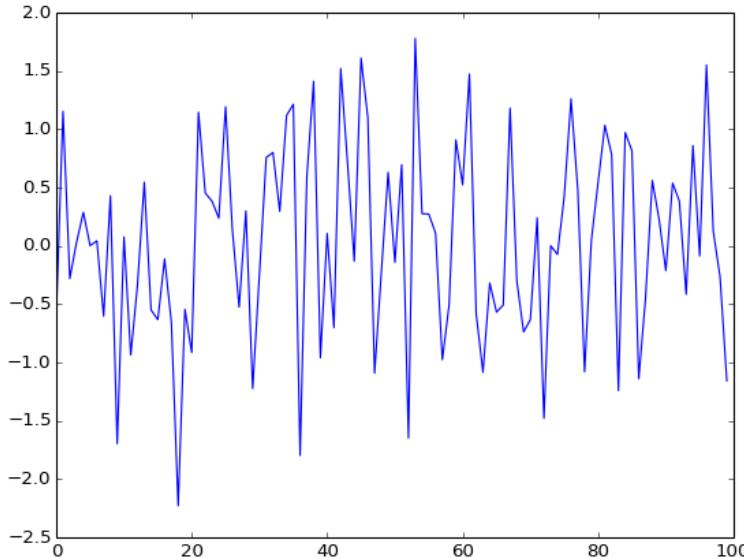
The manual is thoughtfully written but also quite dense (and somewhat evangelical)

The presentation in this and our remaining lectures is more of a tutorial style based around examples

### Example: Plotting a White Noise Process

To begin, let's suppose that 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:



This is straightforward using the PyPlot library we installed earlier

```
using PyPlot
ts_length = 100
epsilon_values = randn(ts_length)
plot(epsilon_values, "b-")
```

You should be able to run that code either in IJulia or in the standard REPL (the basic interpreter)

In brief,

- `using PyPlot` makes the functionality in PyPlot available for use
  - In particular, it pulls the names exported by the PyPlot module into the global scope
  - One of these is `plot()`, which in turn calls the `plot` function from Matplotlib
- `randn()` is a Julia function from the standard library for generating standard normals

**Importing Functions** The effect of the statement `using PyPlot` is to make all the names exported by the PyPlot module available in the global scope

If you prefer to be more selective you can replace `using PyPlot` with `import PyPlot: plot`

Now only the `plot` function is accessible

Since our program uses only the plot function from this module, either would have worked in the previous example

**Arrays** The function call `epsilon_values = randn(ts_length)` creates one of the most fundamental Julia data types: an array

```
julia> typeof(epsilon_values)
Array{Float64,1}

julia> epsilon_values
100-element Array{Float64,1}:
 -0.908823
 -0.759142
 -1.42078
  0.792799
  0.577181
  1.74219
 -0.912529
  1.06259
  0.5766
 -0.0172788
 -0.591671
 -1.02792
 ...
 -1.29412
 -1.12475
  0.437858
 -0.709243
 -1.96053
  1.31092
  1.19819
  1.54028
 -0.246204
 -1.23305
 -1.16484
```

The information from `typeof()` tells us that `epsilon_values` is an array of 64 bit floating point values, of dimension 1

Julia arrays are quite flexible — they can store heterogeneous data for example

```
julia> x = [10, "foo", false]
3-element Array{Any,1}:
 10
 "foo"
 false
```

Notice now that the data type is recorded as `Any`, since the array contains mixed data

The first element of `x` is an integer

```
julia> typeof(x[1])
Int64
```

The second is a string

```
julia> typeof(x[2])
ASCIIString (constructor with 2 methods)
```

The third is the boolean value `false`

```
julia> typeof(x[3])
Bool
```

Notice from the above that

- array indices start at 1 (unlike Python, where arrays are zero-based)
- array elements are referenced using square brackets (unlike MATLAB and Fortran)

Julia contains many functions for acting on arrays — we'll review them later

For now here's several examples, applied to the same list `x = [10, "foo", false]`

```
julia> length(x)
3

julia> pop!(x)
false

julia> x
2-element Array{Any,1}:
 10
 "foo"

julia> push!(x, "bar")
3-element Array{Any,1}:
 10
 "foo"
 "bar"

julia> x
3-element Array{Any,1}:
 10
 "foo"
 "bar"
```

The first example just returns the length of the list

The second, `pop!()`, pops the last element off the list and returns it

In doing so it changes the list (by dropping the last element)

Because of this we call `pop!` a **mutating method**

It's conventional in Julia that mutating methods end in `!` to remind the user that the function has other effects beyond just returning a value

The function `push!()` is similar, expect that it appends its second argument to the array

**For Loops** Although there's no need in terms of what we wanted to achieve with our program, for the sake of learning syntax let's rewrite our program to use a for loop

```
using PyPlot
ts_length = 100
epsilon_values = Array(Float64, ts_length)
for i in 1:ts_length
    epsilon_values[i] = randn()
end
plot(epsilon_values, "b-")
```

Here we first declared `epsilon_values` to be an empty array for storing 64 bit floating point numbers

The `for` loop then populates this array by successive calls to `randn()`

- Called without an argument, `randn()` returns a single float

Like all code blocks in Julia, the end of the for loop code block (which is just one line here) is indicated by the keyword `end`

The word `in` from the for loop can be replaced by symbol `=`

The expression `1:ts_length` creates an **iterator** that is looped over — in this case the integers from 1 to `ts_length`

Iterators are memory efficient because the elements are generated on the fly rather than stored in memory

In Julia you can also loop directly over arrays themselves, like so

```
words = ["foo", "bar"]
for word in words
    println("Hello $word")
end
```

The output is

```
Hello foo
Hello bar
```

**While Loops** The syntax for the while loop contains no surprises

```
using PyPlot
ts_length = 100
epsilon_values = Array(Float64, ts_length)
i = 1
while i <= ts_length
    epsilon_values[i] = randn()
    i = i + 1
end
plot(epsilon_values, "b-")
```

The next example does the same thing with a condition and the `break` statement

```
using PyPlot
ts_length = 100
epsilon_values = Array(Float64, ts_length)
i = 1
while true
    epsilon_values[i] = randn()
    i = i + 1
    if i == ts_length
        break
    end
end
plot(epsilon_values, "b-")
```

**User-Defined Functions** For the sake of the exercise, let's now go back to the `for` loop but re-structure our program so that generation of random variables takes place within a user-defined function

```
using PyPlot

function generate_data(n)
    epsilon_values = Array(Float64, n)
    for i = 1:n
        epsilon_values[i] = randn()
    end
    return epsilon_values
end

ts_length = 100
data = generate_data(ts_length)
plot(data, "b-")
```

Here

- `function` is a Julia keyword that indicates the start of a function definition
- `generate_data` is an arbitrary name for the function
- `return` is a keyword indicating the return value

**A Slightly More Useful Function** Of course the function `generate_data` is completely contrived. We could just write the following and be done

```
ts_length = 100
data = randn(ts_length)
plot(data, "b-")
```

Let's make a slightly more useful function

This function will be passed a choice of probability distribution and respond by plotting a histogram of observations

In doing so we'll make use of the Distributions package

```
julia> Pkg.add("Distributions")
```

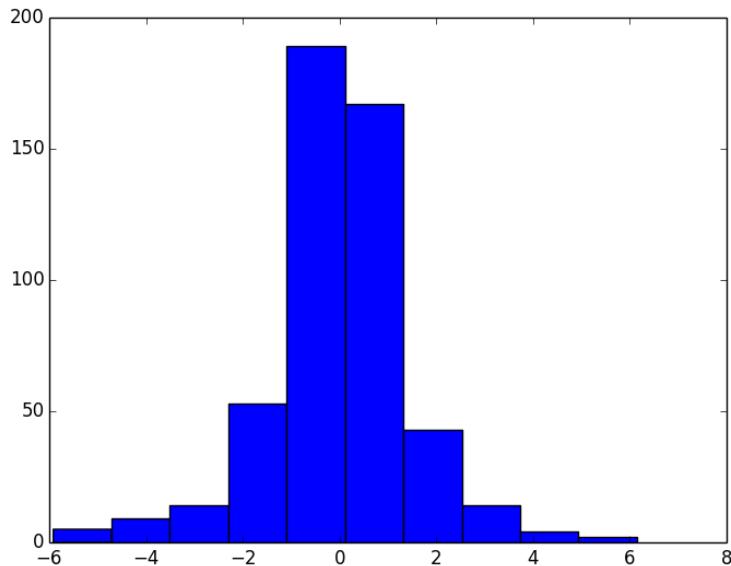
Here's the code

```
using PyPlot
using Distributions

function plot_histogram(distribution, n)
    epsilon_values = rand(distribution, n) # n draws from distribution
    PyPlot.plt.hist(epsilon_values)
end

lp = Laplace()
plot_histogram(lp, 500)
```

The resulting figure looks like this



Let's have a casual discussion of how all this works while leaving technical details for later in the lectures

First, `lp = Laplace()` creates an instance of a data type defined in the `Distributions` module that represents the Laplace distribution

The name `lp` is bound to this object

When we make the function call `plot_histogram(lp, 500)` the code in the body of the function `plot_histogram` is run with

- the name `distribution` bound to the same object as `lp`
- the name `n` bound to the integer 500

**A Mystery** Now consider the function call `rand(distribution, n)`

This looks like something of a mystery

The function `rand()` is defined in the base library such that `rand(n)` returns `n` uniform random variables on  $[0, 1]$

```
julia> rand(3)
3-element Array{Float64,1}:
 0.856817
 0.981502
 0.510947
```

On the other hand, `distribution` points to a data type representing the Laplace distribution that has been defined in a third party package

So how can it be that `rand()` is able to take this kind of object as an argument and return the output that we want?

The answer in a nutshell is **multiple dispatch**

This refers to the idea that functions in Julia can have different behavior depending on the particular arguments that they're passed

Hence in Julia we can take an existing function and give it a new behavior by defining how it acts on a new type of object

The interpreter knows which function definition to apply in a given setting by looking at the types of the objects the function is called on

In Julia these alternative versions of a function are called **methods**

**A Small Problem** In many situations multiple dispatch provides a clean solution for resolving the correct action for a given function in a given setting

You can see however that caution is sometimes required by looking at the line `PyPlot.plt.hist(epsilon_values)` from the code above

A function called `hist()` exists in the standard library and is always available

```
julia> hist([5, 10, 15, 20])
(0.0:5.0:20.0, [1, 1, 1, 1])
```

In addition, to maintain unified syntax with Matplotlib, the library PyPlot also defines its own version of `hist()`, for plotting

Because both versions act on arrays, if we simply write `hist(epsilon_values)` the interpreter can't tell which version to invoke

In fact in this case it falls back to the first one defined, which is not the one defined by PyPlot

This is the reason we need to be more specific, writing `PyPlot.plt.hist(epsilon_values)` instead of just `hist(epsilon_values)`

### Exercises

**Exercise 1** Recall that  $n!$  is read as “ $n$  factorial” and defined as  $n! = n \times (n - 1) \times \cdots \times 2 \times 1$

In Julia you can compute this value with `factorial(n)`

Write your own version of this function, called `factorial2`, using a for loop

**Exercise 2** The binomial random variable  $Y \sim Bin(n, p)$  represents

- number of successes in  $n$  binary trials
- each trial succeeds with probability  $p$

Using only `rand()` from the set of Julia’s built in random number generators (not the Distributions package), 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

For random number generation use only `rand()`

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 falls 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

Once again use only `rand()` as your random number generator

**Exercise 5** 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

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

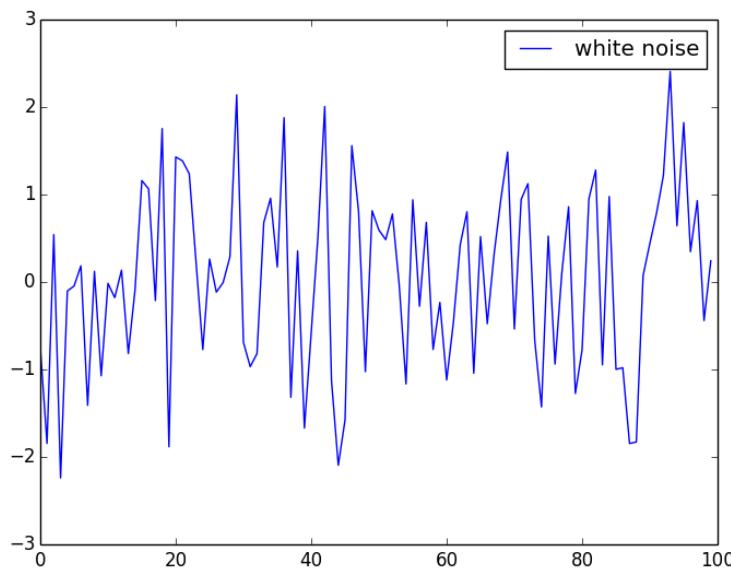
**Exercise 6** To do the next exercise, you will need to know how to produce a plot legend

The following example should be sufficient to convey the idea

```
using PyPlot

x = randn(100)
plot(x, "b-", label="white noise")
legend()
```

Running it produces a figure like so



Now, 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

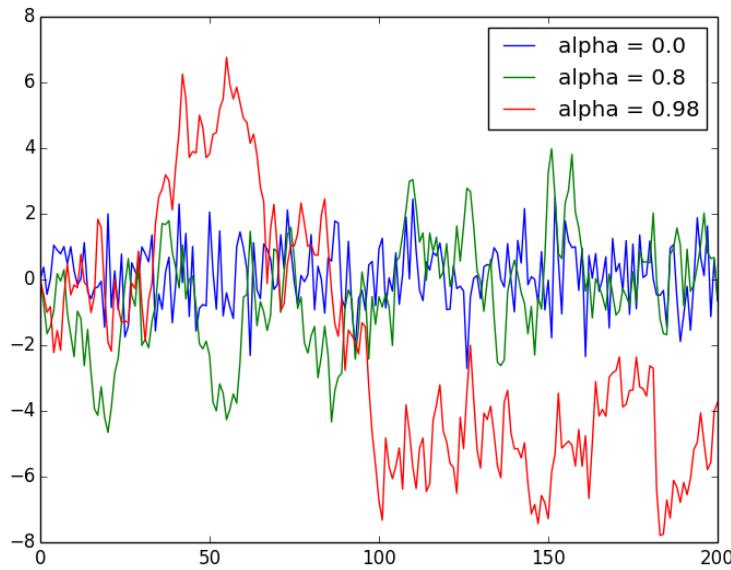
(The figure illustrates how time series with the same one-step-ahead conditional volatilities, as these three processes have, can have very different unconditional volatilities.)

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
- If you omit the argument "`b-`" to the `plot` function, PyPlot will automatically select different colors for each line

## Solutions

[Solution notebook](#)



## 1.3 Julia Essentials

### Contents

- *Julia Essentials*
  - Overview
  - Common Data Types
  - Input and Output
  - Iterating
  - Comparisons and Logical Operators
  - User Defined Functions
  - Exercises
  - Solutions

Having covered a few examples, let's now turn to a more systematic exposition of the essential features of the language

### Overview

Topics:

- Common data types
- Basic file I/O
- Iteration
- More on user-defined functions

- Comparisons and logic

### Common Data Types

Like most languages, Julia language defines and provides functions for operating on standard data types such as

- integers
- floats
- strings
- arrays, etc...

Let's learn a bit more about them

**Primitive Data Types** A particularly simple data type is a Boolean value, which can be either `true` or `false`

```
julia> x = true
true

julia> typeof(x)
Bool

julia> y = 1 > 2  # Now y = false
false
```

Under addition, `true` is converted to 1 and `false` is converted to 0

```
julia> true + false
1

julia> sum([true, false, false, true])
2
```

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

(Computers distinguish between floats and integers because arithmetic is handled in a different way)

```
julia> typeof(1.0)
Float64

julia> typeof(1)
Int64
```

If you're running a 32 bit system you'll still see `Float64`, but you will see `Int32` instead of `Int64` (see the section on Integer types from the Julia manual)

Arithmetic operations are fairly standard

```
julia> x = 2; y = 1.0
1.0

julia> x * y
2.0

julia> x^2
4

julia> y / x
0.5
```

Although the `*` can be omitted for multiplication between variables and numeric literals

```
julia> 2x - 3y
1.0
```

Also, you can use function (instead of infix) notation if you so desire

```
julia> +(10, 20)
30

julia> *(10, 20)
200
```

Complex numbers are another primitive data type, with the imaginary part being specified by `im`

```
julia> x = 1 + 2im
1 + 2im

julia> y = 1 - 2im
1 - 2im

julia> x * y  # Complex multiplication
5 + 0im
```

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

**Strings** A string is a data type for storing a sequence of characters

```
julia> x = "foobar"
"foobar"

julia> typeof(x)
ASCIIString (constructor with 2 methods)
```

You've already seen examples of Julia's simple string formatting operations

```
julia> x = 10; y = 20
20

julia> "x = $x"
"x = 10"
```

```
julia> "x + y = $(x + y)"
"x + y = 30"
```

To concatenate strings use \*

```
julia> "foo" * "bar"
"foobar"
```

Julia provides many functions for working with strings

```
julia> s = "Charlie don't surf"
"Charlie don't surf"

julia> split(s)
3-element Array{SubString{ASCIIString},1}:
 "Charlie"
 "don't"
 "surf"

julia> replace(s, "surf", "ski")
"Charlie don't ski"

julia> split("fee,fi,fo", ",")
3-element Array{SubString{ASCIIString},1}:
 "fee"
 "fi"
 "fo"

julia> strip(" foobar ")  # Remove whitespace
"foobar"
```

Julia can also find and replace using regular expressions (see the documentation on regular expressions for more info)

```
julia> match(r"(\d+)", "Top 10")  # Find numerals in string
RegexMatch("10", 1="10")
```

**Containers** Julia has several basic types for storing collections of data

We have already discussed arrays

A related data type is **tuples**, which can act like “immutable” arrays

```
julia> x = ("foo", "bar")
("foo", "bar")

julia> typeof(x)
(ASCIIString, ASCIIString)
```

An immutable object is one that cannot be altered once it resides in memory

In particular, tuples do not support item assignment:

```
julia> x[1] = 42
ERROR: `setindex!` has no method matching setindex!(::(ASCIIString,ASCIIString), ::Int64, ::Int64)
```

This is similar to Python, as is the fact that the parenthesis can be omitted

```
julia> x = "foo", "bar"
("foo", "bar")
```

Another similarity with Python is tuple unpacking, which means that the following convenient syntax is valid

```
julia> x = ("foo", "bar")
("foo", "bar")

julia> word1, word2 = x
("foo", "bar")

julia> word1
"foo"

julia> word2
"bar"
```

**Referencing Items** The last element of a sequence type can be accessed with the keyword `end`

```
julia> x = [10, 20, 30, 40]
4-element Array{Int64,1}:
 10
 20
 30
 40

julia> x[end]
40

julia> x[end-1]
30
```

To access multiple elements of an array or tuple, you can use slice notation

```
julia> x[1:3]
3-element Array{Int64,1}:
 10
 20
 30

julia> x[2:end]
3-element Array{Int64,1}:
 20
 30
 40
```

The same slice notation works on strings

```
julia> "foobar"[3:end]
"obar"
```

**Dictionaries** Another container type worth mentioning is dictionaries

Dictionaries are like arrays except that the items are named instead of numbered

```
julia> d = {"name" => "Frodo", "age" => 33}
Dict{Any,Any} with 2 entries:
  "name" => "Frodo"
  "age"   => 33

julia> d["age"]
33
```

The strings `name` and `age` are called the **keys**

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

They can be accessed via `keys(d)` and `values(d)` respectively

## Input and Output

Let's have a quick look at reading from and writing to text files

We'll start with writing

```
julia> f = open("newfile.txt", "w") # "w" for writing
IOStream(<file newfile.txt>)

julia> write(f, "testing\n")      # \n for newline
7

julia> write(f, "more testing\n")
12

julia> close(f)
```

The effect of this is to create a file called `newfile.txt` in your present working directory with contents

```
testing
more testing
```

We can read the contents of `newline.txt` as follows

```
julia> f = open("newfile.txt", "r") # Open for reading
IOStream(<file newfile.txt>)

julia> print(readall(f))
testing
more testing
```

```
julia> close(f)
```

Often when reading from a file we want to step through the lines of a file, performing an action on each one

There's a neat interface to this in Julia, which takes us to our next topic

## Iterating

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

Julia's provides neat, flexible tools for iteration as we now discuss

**Iterables** An iterable is something you can put on the right hand side of `for` and loop over

These include sequence data types like arrays

```
actions = ["surf", "ski"]
for action in actions
    println("Charlie don't $action")
end
```

They also include so-called **iterators**

You've already come across these types of objects

```
julia> for i in 1:3 print(i) end
123
```

If you ask for the keys of dictionary you get an iterator

```
julia> d = {"name" => "Frodo", "age" => 33}
Dict{Any,Any} with 2 entries:
"name" => "Frodo"
"age"  => 33

julia> keys(d)
KeyIterator for a Dict{Any,Any} with 2 entries. Keys:
"name"
"age"
```

This makes sense, since the most common thing you want to do with keys is loop over them

The benefit of providing an iterator rather than an array, say, is that the former is more memory efficient

Should you need to transform an iterator into an array you can always use `collect()`

```
julia> collect(keys(d))
2-element Array{Any,1}:
 "name"
 "age"
```

**Looping without Indices** You can loop over sequences without explicit indexing, which often leads to neater code

For example compare

```
for x in x_values
    println(x * x)
end
```

with

```
for i in 1:length(x_values)
    println(x_values[i] * x_values[i])
end
```

Julia provides some functional-style helper functions (similar to Python) to facilitate 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)
    println("The capital of $country is $city")
end
```

If we happen to need the index as well as the value, one option is to use `enumerate()`

The following snippet will give you the idea

```
countries = ("Japan", "Korea", "China")
cities = ("Tokyo", "Seoul", "Beijing")
for (i, country) in enumerate(countries)
    city = cities[i]
    println("The capital of $country is $city")
end
```

**Comprehensions** Comprehensions are an elegant tool for creating new arrays or dictionaries from iterables

Here's some examples

```
julia> doubles = [2i for i in 1:4]
4-element Array{Int64,1}:
 2
 4
 6
 8

julia> animals = ["dog", "cat", "bird"]
3-element Array{ASCIIString,1}:
 "dog"
```

```

"cat"
"bird"

julia> plurals = [animal * "s" for animal in animals]
3-element Array{Union(ASCIIString,UTF8String),1}:
"dogs"
"cats"
"birds"

julia> [i + j for i=1:3, j=4:6]  # can specify multiple parameters
3x3 Array{Int64,2}:
5 6 7
6 7 8
7 8 9

julia> [i + j for i=1:3, j=4:6, k=7:9]
3x3x3 Array{Int64,3}:
[:, :, 1] =
5 6 7
6 7 8
7 8 9

[:, :, 2] =
5 6 7
6 7 8
7 8 9

[:, :, 3] =
5 6 7
6 7 8
7 8 9

```

The same kind of expression works for dictionaries

```

julia> d = {"$i" => i for i in 1:3}
Dict{Any,Any} with 3 entries:
"1" => 1
"2" => 2
"3" => 3

```

## Comparisons and Logical Operators

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

```

julia> x = 1
1

julia> x == 2
false

```

For “not equal” use !=

```
julia> x != 3
true
```

In Julia we can *chain* inequalities

```
julia> 1 < 2 < 3
true

julia> 1 <= 2 <= 3
true
```

In many languages you can use integers or other values when testing conditions but Julia is more fussy

```
julia> while 0 println("foo") end
ERROR: type: non-boolean (Int64) used in boolean context
      in anonymous at no file

julia> if 1 println("foo") end
ERROR: type: non-boolean (Int64) used in boolean context
```

**Combining Expressions** Here are the standard logical connectives (conjunction, disjunction)

```
julia> true && false
false

julia> true || false
true
```

Remember

- P `&&` Q is true if both are true, otherwise it's false
- P `||` Q is false if both are false, otherwise it's true

## User Defined Functions

Let's talk a little more about user defined 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 always a bad idea)

Julia functions are convenient:

- Any number of functions can be defined in a given file
- Any “value” 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 value, including functions

We'll see many examples of these structures in the following lectures

For now let's just cover some of the different ways of defining functions

**Return Statement** In Julia, the `return` statement is optional, so that the following functions have identical behavior

```
function f1(a, b)
    return a * b
end

function f2(a, b)
    a * b
end
```

When no `return` statement is present, the last value obtained when executing the code block is returned

Although some prefer the second option, we often favor the former on the basis that explicit is better than implicit

A function can have arbitrarily many `return` statements, with execution terminating when the first `return` is hit

You can see this in action when experimenting with the following function

```
function foo(x)
    if x > 0
        return "positive"
    end
    return "nonpositive"
end
```

**Other Syntax for Defining Functions** For short function definitions Julia offers some attractive simplified syntax

First, when the function body is a simple expression, it can be defined without the `function` keyword or `end`

```
julia> f(x) = sin(1 / x)
f (generic function with 2 methods)
```

Let's check that it works

```
julia> f(1 / pi)
1.2246467991473532e-16
```

Julia also allows for you to define anonymous functions

For example, to define `f(x) = sin(1 / x)` you can use `x -> sin(1 / x)`

The difference is that the second function has no name bound to it

How can you use a function with no name?

Typically it's as an argument to another function

```
julia> map(x -> sin(1 / x), randn(3)) # Apply function to each element
3-element Array{Float64,1}:
 0.744193
-0.370506
-0.458826
```

**Optional and Keyword Arguments** Function arguments can be given default values

```
function f(x, a=1)
    return exp(cos(a * x))
end
```

If the argument is not supplied the default value is substituted

```
julia> f(pi)
0.36787944117144233

julia> f(pi, 2)
2.718281828459045
```

Another option is to use **keyword** arguments

The difference between keyword and standard (positional) arguments is that they are parsed and bound by name rather than order in the function call

For example, in the call

```
simulate(param1, param2, max_iterations=100, error_tolerance=0.01)
```

the last two arguments are keyword arguments and their order is irrelevant (as long as they come after the positional arguments)

To define a function with keyword arguments you need to use ; like so

```
function simulate(param1, param2; max_iterations=100, error_tolerance=0.01)
    # Function body here
end
```

## Exercises

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

Part 2: Using a comprehension, count the number of even numbers in 0,...,99

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

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

**Exercise 2** Consider the polynomial

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

Using `enumerate()` in your loop, write a function `p` such that `p(x, coeff)` computes the value in (1.1) given a point `x` and an array of coefficients `coeff`

**Exercise 3** Write a function that takes a string as an argument and returns the number of capital letters in the string

Hint: `uppercase("foo")` 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 an array, tuple or string

**Exercise 5** The Julia libraries include functions for interpolation and approximation

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

In particular, 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[1] < point[2] < ... < point[n] = b`

Aim for clarity, not efficiency

**Exercise 6** The following data lists US cities and their populations

```
new york: 8244910
los angeles: 3819702
chicago: 2707120
houston: 2145146
philadelphia: 1536471
phoenix: 1469471
san antonio: 1359758
san diego: 1326179
dallas: 1223229
```

Copy this text into a text file called `us_cities.txt` and save it in your present working directory

- That is, save it in the location Julia returns when you call `pwd()`

Write a program to calculate total population across these cities

Hints:

- If `f` is a file object then `eachline(f)` provides an iterable that steps you through the lines in the file
- `int("100")` converts the string "100" into an integer

## Solutions

[Solution notebook](#)

## 1.4 Vectors, Arrays and Matrices

### Contents

- *Vectors, Arrays and Matrices*
  - *Overview*
  - *Array Basics*
  - *Operations on Arrays*
  - *Linear Algebra*
  - *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

In Julia, arrays are the most important data type for working with collections of numerical data

In this lecture we give more details on

- creating and manipulating Julia arrays
- fundamental array processing operations
- basic matrix algebra

### Array Basics

**Shape and Dimension** We've already seen some Julia arrays in action

```
julia> a = [10, 20, 30]
3-element Array{Int64,1}:
 10
 20
 30

julia> a = ["foo", "bar", 10]
3-element Array{Any,1}:
 "foo"
 "bar"
 10
```

The REPL tells us that the arrays are of types `Array{Int64,1}` and `Array{Any,1}` respectively

Here `Int64` and `Any` are types for the elements inferred by the compiler

We'll talk more about types later on

The 1 in `Array{Int64,1}` and `Array{Any,1}` indicates that the array is *one dimensional*

This is the default for many Julia functions that create arrays

```
julia> typeof(linspace(0, 1, 100))
Array{Float64,1}

julia> typeof(randn(100))
Array{Float64,1}
```

To say that an array is one dimensional is to say that it is flat — neither a row nor a column vector

We can also confirm that `a` is flat using the `size()` or `ndims()` functions

```
julia> size(a)
(3,)

julia> ndims(a)
1
```

The syntax `(3,)` displays a tuple containing one element

Here it gives the size along the one dimension that exists

Here's a function that creates a two-dimensional array

```
julia> eye(3)
3x3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> diagm([2, 4])
2x2 Array{Int64,2}:
 2  0
 0  4
```

```
julia> size(eye(3))
(3,3)
```

**Array vs Vector vs Matrix** In Julia, in addition to arrays you will see the types `Vector` and `Matrix`. However, these are just aliases for one- and two-dimensional arrays respectively

```
julia> Array{Int64, 1} == Vector{Int64}
true

julia> Array{Int64, 2} == Matrix{Int64}
true

julia> Array{Int64, 1} == Matrix{Int64}
false

julia> Array{Int64, 3} == Matrix{Int64}
false
```

The only slightly disturbing thing here is that the common mathematical terms “row vector” and “column vector” don’t make sense in Julia

By definition, a `Vector` in Julia is flat and hence neither row nor column

**Changing Dimensions** The primary function for changing the dimension of an array is `reshape()`

```
julia> a = [10, 20, 30, 40]
4-element Array{Int64,1}:
 10
 20
 30
 40

julia> reshape(a, 2, 2)
2x2 Array{Int64,2}:
 10  30
 20  40

julia> reshape(a, 1, 4)
1x4 Array{Int64,2}:
 10  20  30  40
```

Notice that this function returns a new copy of the reshaped array rather than modifying the existing one

To collapse an array along one dimension you can use `squeeze()`

```
julia> a = [1 2 3 4] # Two dimensional
1x4 Array{Int64,2}:
 1  2  3  4
```

```
julia> squeeze(a, 1)
4-element Array{Int64,1}:
 1
 2
 3
 4
```

The return value is an Array with the specified dimension “flattened”

**Why Vectors?** As we’ve seen, in Julia we have both

- one-dimensional arrays (flat arrays, or vectors)
- two-dimensional arrays of dimension (1, n) or (n, 1) containing the same elements

Why do we need both?

On one hand, dimension matters when we come to matrix algebra

- Multiplying by a row vector is different to multiplication by a column vector

However, if our vectors are *not* multiplying matrices, their dimensions don’t matter, and hence are an unnecessary complication

This is why many Julia functions return flat arrays by default

## Creating Arrays

**Functions that Return Arrays** We’ve already seen some functions for creating arrays

```
julia> eye(2)
2x2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0

julia> zeros(3)
3-element Array{Float64,1}:
 0.0
 0.0
 0.0
```

You can create an empty array using the `Array()` constructor

```
julia> x = Array(Float64, 2, 2)
2x2 Array{Float64,2}:
 0.0           2.82622e-316
 2.76235e-318  2.82622e-316
```

The printed values you see here are just garbage values

(the existing contents of the allocated memory slots being interpreted as 64 bit floats)

Other important functions that return arrays are

```
julia> ones(2, 2)
2x2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0

julia> fill("foo", 2, 2)
2x2 Array{ASCIIString,2}:
 "foo"  "foo"
 "foo"  "foo"
```

**Manual Array Definitions** As we've seen, you can create one dimensional arrays from manually specified data like so

```
julia> a = [10, 20, 30, 40]
4-element Array{Int64,1}:
 10
 20
 30
 40
```

In two dimensions we can proceed as follows

```
julia> a = [10 20 30 40] # Two dimensional, shape is 1 x n
1x4 Array{Int64,2}:
 10  20  30  40

julia> ndims(a)
2

julia> a = [10 20; 30 40] # 2 x 2
2x2 Array{Int64,2}:
 10  20
 30  40
```

You might then assume that `a = [10; 20; 30; 40]` creates a two dimensional column vector by unfortunately this isn't the case

```
julia> a = [10; 20; 30; 40]
4-element Array{Int64,1}:
 10
 20
 30
 40

julia> ndims(a)
1
```

Instead transpose the row vector

```
julia> a = [10 20 30 40] '
4x1 Array{Int64,2}:
 10
```

```

20
30
40

julia> ndims(a)
2

```

**Array Indexing** We've already seen the basics of array indexing

```

julia> a = collect(10:10:40)
4-element Array{Int64,1}:
 10
 20
 30
 40

julia> a[end-1]
30

julia> a[1:3]
3-element Array{Int64,1}:
 10
 20
 30

```

For 2D arrays the index syntax is straightforward

```

julia> a = randn(2, 2)
2x2 Array{Float64,2}:
 1.37556  0.924224
 1.52899  0.815694

julia> a[1, 1]
1.375559922478634

julia> a[1, :]
1x2 Array{Float64,2}:
 1.37556  0.924224

julia> a[:, 1]  # First column
2-element Array{Float64,1}:
 1.37556
 1.52899

```

Booleans can be used to extract elements

```

julia> a = randn(2, 2)
2x2 Array{Float64,2}:
 -0.121311  0.654559
 -0.297859  0.89208

julia> b = [true false; false true]
2x2 Array{Bool,2}:

```

```

true  false
false   true

julia> a[b]
2-element Array{Float64,1}:
 -0.121311
  0.89208

```

This is useful for conditional extraction, as we'll see below

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

```

julia> a = Array(Float64, 4)
4-element Array{Float64,1}:
 1.30822e-282
 1.2732e-313
 4.48229e-316
 1.30824e-282

julia> a[2:end] = 42
42

julia> a
4-element Array{Float64,1}:
 1.30822e-282
 42.0
 42.0
 42.0

```

**Passing Arrays** As in Python, all arrays are passed by reference

What this means is that if `a` is an array and we set `b = a` then `a` and `b` point to exactly the same data

Hence any change in `b` is reflected in `a`

```

julia> a = ones(3)
3-element Array{Float64,1}:
 1.0
 1.0
 1.0

julia> b = a
3-element Array{Float64,1}:
 1.0
 1.0
 1.0

julia> b[3] = 44
44

julia> a
3-element Array{Float64,1}:
 1.0
 1.0
 44.0

```

```
1.0
1.0
44.0
```

If you are a MATLAB programmer perhaps you are recoiling in horror at this idea

But this is actually the more sensible default – after all, it's very inefficient to copy arrays unnecessarily

If you do need an actual copy in Julia, just use `copy()`

```
julia> a = ones(3)
3-element Array{Float64,1}:
 1.0
 1.0
 1.0

julia> b = copy(a)
3-element Array{Float64,1}:
 1.0
 1.0
 1.0

julia> b[3] = 44
44

julia> a
3-element Array{Float64,1}:
 1.0
 1.0
 1.0
```

## Operations on Arrays

**Array Methods** Julia provides standard functions for acting on arrays, some of which we've already seen

```
julia> a = [-1, 0, 1]
3-element Array{Int64,1}:
 -1
  0
  1

julia> length(a)
3

julia> sum(a)
0

julia> mean(a)
0.0
```

```
julia> std(a)
1.0

julia> var(a)
1.0

julia> maximum(a)
1

julia> minimum(a)
-1

julia> b = sort(a, rev=true)    # Returns new array, original not modified
3-element Array{Int64,1}:
 1
 0
 -1

julia> b === a    # === tests if arrays are identical (i.e share same memory)
false

julia> b = sort!(a, rev=true)   # Returns *modified original* array
3-element Array{Int64,1}:
 1
 0
 -1

julia> b === a
true
```

**Matrix Algebra** For two dimensional arrays, `*` means matrix multiplication

```
julia> a = ones(1, 2)
1x2 Array{Float64,2}:
 1.0  1.0

julia> b = ones(2, 2)
2x2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0

julia> a * b
1x2 Array{Float64,2}:
 2.0  2.0

julia> b * a'
2x1 Array{Float64,2}:
 2.0
 2.0
```

To solve the linear system  $A \cdot X = B$  for  $X$  use  $A \setminus B$

```
julia> A = [1 2; 2 3]
2x2 Array{Int64,2}:
 1  2
 2  3

julia> B = ones(2, 2)
2x2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0

julia> A \ B
2x2 Array{Float64,2}:
 -1.0  -1.0
  1.0   1.0

julia> inv(A) * B
2x2 Array{Float64,2}:
 -1.0  -1.0
  1.0   1.0
```

Although the last two operations give the same result, the first one is numerically more stable and should be preferred in most cases

Multiplying two **one** dimensional vectors gives an error — which is reasonable since the meaning is ambiguous

```
julia> ones(2) * ones(2)
ERROR: `*` has no method matching *(::Array{Float64,1}, ::Array{Float64,1})
```

If you want an inner product in this setting use dot()

```
julia> dot(ones(2), ones(2))
2.0
```

Matrix multiplication using one dimensional vectors is a bit inconsistent — pre-multiplication by the matrix is OK, but post-multiplication gives an error

```
julia> b = ones(2, 2)
2x2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0

julia> b * ones(2)
2-element Array{Float64,1}:
 2.0
 2.0

julia> ones(2) * b
ERROR: DimensionMismatch("*")
in gemm_wrapper! at linalg/matmul.jl:275
in * at linalg/matmul.jl:74
```

It's probably best to give your vectors dimension before you multiply them against matrices

## Elementwise Operations

**Algebraic Operations** Suppose that we wish to multiply every element of matrix A with the corresponding element of matrix B

In that case we need to replace \* (matrix multiplication) with .\* (elementwise multiplication)

For example, compare

```
julia> ones(2, 2) * ones(2, 2)    # Matrix multiplication
2x2 Array{Float64,2}:
 2.0  2.0
 2.0  2.0

julia> ones(2, 2) .* ones(2, 2)    # Element by element multiplication
2x2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0
```

This is a general principle: .x means apply operator x elementwise

```
julia> A = -ones(2, 2)
2x2 Array{Float64,2}:
 -1.0  -1.0
 -1.0  -1.0

julia> A.^2    # Square every element
2x2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0
```

However in practice some operations are unambiguous and hence the . can be omitted

```
julia> ones(2, 2) + ones(2, 2)    # Same as ones(2, 2) .+ ones(2, 2)
2x2 Array{Float64,2}:
 2.0  2.0
 2.0  2.0
```

Scalar multiplication is similar

```
julia> A = ones(2, 2)
2x2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0

julia> 2 * A    # Same as 2 .* A
2x2 Array{Float64,2}:
 2.0  2.0
 2.0  2.0
```

In fact you can omit the \* altogether and just write 2A

**Elementwise Comparisons** Elementwise comparisons also use the .x style notation

```
julia> a = [10, 20, 30]
3-element Array{Int64,1}:
 10
 20
 30

julia> b = [-100, 0, 100]
3-element Array{Int64,1}:
 -100
  0
 100

julia> b .> a
3-element BitArray{1}:
 false
 false
 true

julia> a .== b
3-element BitArray{1}:
 false
 false
 false
```

We can also do comparisons against scalars with parallel syntax

```
julia> b
3-element Array{Int64,1}:
 -100
  0
 100

julia> b .> 1
3-element BitArray{1}:
 false
 false
 true
```

This is particularly useful for *conditional extraction* — extracting the elements of an array that satisfy a condition

```
julia> a = randn(4)
4-element Array{Float64,1}:
 0.0636526
 0.933701
 -0.734085
 0.531825

julia> a .< 0
4-element BitArray{1}:
 false
 false
 true
 false
```

```
julia> a[a .< 0]
1-element Array{Float64,1}:
 -0.734085
```

**Vectorized Functions** Julia provides standard mathematical functions such as `log`, `exp`, `sin`, etc.

```
julia> log(1.0)
0.0
```

By default, these functions act *elementwise* on arrays

```
julia> log(ones(4))
4-element Array{Float64,1}:
 0.0
 0.0
 0.0
 0.0
```

Functions that act elementwise on arrays in this manner are called **vectorized functions**

Note that we can get the same result as with a comprehension or more explicit loop

```
julia> [log(x) for x in ones(4)]
4-element Array{Float64,1}:
 0.0
 0.0
 0.0
 0.0
```

In Julia loops are typically fast and hence the need for vectorized functions is less intense than for some other high level languages

Nonetheless the syntax is convenient

## Linear Algebra

Julia provides some a great deal of additional functionality related to linear operations

```
julia> A = [1 2; 3 4]
2x2 Array{Int64,2}:
 1  2
 3  4

julia> det(A)
-2.0

julia> trace(A)
5

julia> eigvals(A)
2-element Array{Float64,1}:
 -0.372281
```

```
5.37228
julia> rank(A)
2
```

For more details see the [linear algebra section](#) of the standard library

### Exercises

**Exercise 1** Consider the stochastic difference equation

$$X_{t+1} = AX_t + b + \Sigma W_t \quad (1.2)$$

Here  $\{W_t\}$  is an iid vector of shocks with variance-covariance matrix equal to the identity matrix

Letting  $S_t$  denote the variance of  $X_t$  and using the rules for computing variances in matrix expressions, it can be shown from (1.2) that  $\{S_t\}$  obeys

$$S_{t+1} = AS_t A' + \Sigma \Sigma' \quad (1.3)$$

Provided that all eigenvalues of  $A$  lie within the unit circle, the sequence  $\{S_t\}$  converges to a unique limit  $S$

This is the **unconditional variance** or **asymptotic variance** of the stochastic difference equation

As an exercise, try writing a simple function that solves for the limit  $S$  by iterating on (1.3) given  $A$  and  $\Sigma$

To test your solution, observe that the limit  $S$  is a solution to the matrix equation

$$S = ASA' + Q \quad \text{where} \quad Q := \Sigma \Sigma' \quad (1.4)$$

This kind of equation is known as a **discrete time Lyapunov equation**

The [QuantEcon package](#) provides a function called `solve_discrete_lyapunov` that implements a fast “doubling” algorithm to solve this equation

Test your iterative method against `solve_discrete_lyapunov` using matrices

$$A = \begin{bmatrix} 0.8 & -0.2 \\ -0.1 & 0.7 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 0.5 & 0.4 \\ 0.4 & 0.6 \end{bmatrix}$$

### Solutions

[Solution notebook](#)

## 1.5 Types, Methods and Performance

## Contents

- *Types, Methods and Performance*
  - *Overview*
  - *Types*
  - *Defining Types and Methods*
  - *Writing Fast Code*
  - *Exercises*
  - *Solutions*

## Overview

In this lecture we delve more deeply into the structure of Julia, and in particular into

- the concept of types
- building user defined types
- methods and multiple dispatch

These concepts relate to the way that Julia stores and acts on data

While they might be thought of as advanced topics, some understanding is necessary to

1. Read Julia code written by other programmers
2. Write “well organized” Julia code that’s easy to maintain and debug
3. Improve the speed at which your code runs

At the same time, don’t worry about following all the nuances on your first pass

If you return to these topics after doing some programming in Julia they will make more sense

## Types

In Julia all objects (all “values” in memory) have a type, which can be queried using the `typeof()` function

```
julia> x = 42
42

julia> typeof(x)
Int64
```

Note here that the type resides with the object itself, not with the name `x`

The name `x` is just a symbol bound to an object of type `Int64`

Here we *rebind* it to another object, and now `typeof(x)` gives the type of that new object

```
julia> x = 42.0
42.0

julia> typeof(x)
Float64
```

**Common Types** We've already met many of the types defined in the core Julia language and its standard library

For numerical data, the most common types are integers and floats

For those working on a 64 bit machine, the default integers and floats are 64 bits, and are called `Int64` and `Float64` respectively (they would be `Int32` and `Float64` on a 32 bit machine)

There are many other important types, used for arrays, strings, iterators and so on

```
julia> typeof(1 + 1im)
Complex{Int64} (constructor with 1 method)

julia> typeof(linspace(0, 1, 100))
Array{Float64,1}

julia> typeof(eye(2))
Array{Float64,2}

julia> typeof("foo")
ASCIIString (constructor with 2 methods)

julia> typeof(1:10)
UnitRange{Int64} (constructor with 1 method)

julia> typeof('c') # Single character is a *Char*
Char
```

Type is important in Julia because it determines what operations will be performed on the data in a given situation

Moreover, if you try to perform an action that is unexpected for a given type the function call will usually fail

```
julia> 100 + "100"
ERROR: `+` has no method matching +(::Int64, ::ASCIIString)
```

Some languages will try to guess what the programmer wants here and return 200

Julia doesn't — in this sense, Julia is a "strongly typed" language

Type is important and it's up to the user to supply data in the correct form (as specified by type)

**Methods and Multiple Dispatch** Looking more closely at how this works brings us to a very important topic concerning Julia's data model — methods and multiple dispatch

Let's look again at the error message

```
julia> 100 + "100"
ERROR: `+` has no method matching +(::Int64, ::ASCIIString)
```

As discussed earlier, the operator `+` is just a function, and we can rewrite that call using functional notation to obtain exactly the same result

```
julia> +(100, "100")
ERROR: `+` has no method matching +(::Int64, ::ASCIIString)
```

Multiplication is similar

```
julia> 100 * "100"
ERROR: `*` has no method matching *(::Int64, ::ASCIIString)

julia> *(100, "100")
ERROR: `*` has no method matching *(::Int64, ::ASCIIString)
```

What the message tells us is that `*(a, b)` doesn't work when `a` is an integer and `b` is a string

In particular, the function `*` has no *matching method*

In essence, a **method** in Julia is a version of a function that acts on a particular tuple of data types

For example, if `a` and `b` are integers then a method for multiplying integers is invoked

```
julia> *(100, 100)
10000
```

On the other hand, if `a` and `b` are strings then a method for string concatenation is invoked

```
julia> *("foo", "bar")
"foobar"
```

In fact we can see the precise methods being invoked by applying `@which`

```
julia> @which *(100, 100)
*(x::Int64,y::Int64) at int.jl:47

julia> @which *("foo", "bar")
*(s::String...) at string.jl:76
```

We can see the same process with other functions and their methods

```
julia> isfinite(1.0) # Call isfinite on a float
true

julia> @which(isfinite(1.0))
isfinite(x::FloatingPoint) at float.jl:213

julia> isfinite(1) # Call isfinite on an integer
true

julia> @which(isfinite(1))
isfinite(x::Integer) at float.jl:215
```

Here `isfinite()` is a *function* with multiple *methods*

It has a method for acting on floating points and another method for acting on integers

In fact it has quite a few methods

```
julia> methods(isfinite) # List the methods of isfinite
# 9 methods for generic function "isfinite":
isfinite(x::Float16) at float16.jl:115
isfinite(x::BigFloat) at mpfr.jl:717
isfinite(x::FloatingPoint) at float.jl:213
isfinite(x::Integer) at float.jl:215
isfinite(x::Real) at float.jl:214
isfinite(z)::Complex{T<:Real}) at complex.jl:41
isfinite{T<:Number}(:AbstractArray{T<:Number,1}) at operators.jl:359
isfinite{T<:Number}(:AbstractArray{T<:Number,2}) at operators.jl:360
isfinite{T<:Number}(:AbstractArray{T<:Number,N}) at operators.jl:362
```

The particular method being invoked depends on the data type on which the function is called

We'll discuss some of the more complicated data types you see towards the end of this list as we go along

**Abstract Types** Looking at the list of methods above you can see references to types that we haven't met before, such as `Real` and `Number`

These two types are examples of what are known in Julia as **abstract types**

Abstract types serve a different purpose to **concrete types** such as `Int64` and `Float64`

To understand what that purpose is, imagine that you want to write a function with two methods, one to handle real numbers and the other for complex numbers

As we know, there are multiple types for real numbers, such as integers and floats

There are even multiple integer and float types, such as `Int32`, `Int64`, `Float32`, `Float64`, etc.

If we want to handle all of these types for real numbers in the same way, it's useful to have a "parent" type called `Real`

Rather than writing a separate method for each concrete type, we can just write one for the abstract `Real` type

In this way, the purpose of abstract types is to serve as a unifying parent type that concrete types can "inherit" from

Indeed, we can see that `Float64`, `Int64`, etc. are **subtypes** of `Real` as follows

```
julia> Float64 <: Real
true

julia> Int64 <: Real
true
```

On the other hand, 64 bit complex numbers are not reals

```
julia> Complex64 <: Real
false
```

They are, however, subtypes of Number

```
julia> Complex64 <: Number
true
```

Number in turn is a subtype of Any, which is a parent of all types

```
julia> Number <: Any
true
```

**Type Hierarchy** In fact the types form a hierarchy, with Any at the top of the tree and the concrete types at the bottom

Note that we never see *instances* of abstract types

For example, if x is a value then `typeof(x)` will never return an abstract type

The point of abstract types is simply to categorize the concrete types (as well as other abstract types that sit below them in the hierarchy)

On the other hand, we cannot subtype concrete types

While we can build subtypes of abstract types we cannot do the same for concrete types

**Multiple Dispatch** We can now be a little bit clearer about what happens when you call a function on given types

Suppose we execute the function call `f(a, b)` where a and b are of concrete types S and T respectively

The Julia interpreter first queries the types of a and b to obtain the tuple (S, T)

It then parses the list of methods belonging to f, searching for a match

If it finds a method matching (S, T) it calls that method

If not, it looks to see whether the pair (S, T) matches any method defined for *immediate parent types*

For example, if S is Float64 and T is Complex64 then the immediate parents are FloatingPoint and Number respectively

```
julia> super(Float64)
FloatingPoint

julia> super(Complex64)
Number
```

Hence the interpreter looks next for a method of the form `f(x::FloatingPoint, y::Number)`

If the interpreter can't find a match in immediate parents (supertypes) it proceeds up the tree, looking at the parents of the last type it checked at each iteration

- If it eventually finds a matching method it invokes that method
- If not, we get an error

This is the process that leads to the error that we saw above:

```
julia> *(100, "100")
ERROR: `*` has no method matching *(::Int64, ::ASCIIString)
```

The procedure of matching data to appropriate methods is called **dispatch**

Because the procedure starts from the concrete types and works upwards, dispatch always invokes the *most specific method* that is available

For example, if you have methods for function `f` that handle

1. (`Float64`, `Int64`) pairs
2. (`Number`, `Number`) pairs

and you call `f` with `f(0.5, 1)` then the first method will be invoked

This makes sense because (hopefully) the first method is designed to work well with exactly this kind of data

The second method is probably more of a “catch all” method that handles other data in a less optimal way

## Defining Types and Methods

Let's look at defining our own methods and data types, including composite data types

**User Defined Methods** It's straightforward to add methods to existing functions or functions you've defined

In either case the process is the same:

- use the standard syntax to define a function of the same name
- but specify the data type for the method in the function signature

For example, we saw above that `+` is just a function with various methods

- recall that `a + b` and `+(a, b)` are equivalent

We saw also that the following call fails because it lacks a matching method

```
julia> +(100, "100")
ERROR: `+` has no method matching +(::Int64, ::ASCIIString)
```

This is sensible behavior, but if you want to change it by defining a method to handle the case in question there's nothing to stop you:

```
julia> +(x::Integer, y::ASCIIString) = x + int(y)
+ (generic function with 126 methods)

julia> +(100, "100")
200

julia> 100 + "100"
200
```

Here's another example, involving a user defined function

We begin with a file called `test.jl` in the present working directory with the following content

```
function f(x)
    println("Generic function invoked")
end

function f(x::Number)
    println("Number method invoked")
end

function f(x::Integer)
    println("Integer method invoked")
end
```

Clearly these methods do nothing more than tell you which method is being invoked

Let's now run this and see how it relates to our discussion of method dispatch above

```
julia> include("test.jl")
f (generic function with 3 methods)

julia> f(3)
Integer method invoked

julia> f(3.0)
Number method invoked

julia> f("foo")
Generic function invoked
```

Since 3 is an `Int64` and `Int64 <: Integer <: Number`, the call `f(3)` proceeds up the tree to `Integer` and invokes `f(x::Integer)`

On the other hand, 3.0 is a `Float64`, which is not a subtype of `Integer`

Hence the call `f(3.0)` continues up to `f(x::Number)`

Finally, `f("foo")` is handled by the generic function, since it is not a subtype of `Number`

**User Defined Types** Most languages have facilities for creating new data types and Julia is no exception

```
julia> type Foo end
julia> foo = Foo()
Foo()

julia> typeof(foo)
Foo (constructor with 1 method)
```

Let's make some observations about this code

First note that to create a new data type we use the keyword `type` followed by the name

- By convention, type names use CamelCase (e.g., `FloatingPoint`, `Array`, `AbstractArray`)

When a new data type is created in this way, the interpreter simultaneously creates a *default constructor* for the data type

This constructor is a function for generating new instances of the data type in question

It has the same name as the data type but uses function call notion — in this case `Foo()`

In the code above, `foo = Foo()` is a call to the default constructor

A new instance of type `Foo` is created and the name `foo` is bound to that instance

Now if we want to we can create methods that act on instances of `Foo`

Just for fun, let's define how to add one `Foo` to another

```
julia> +(x::Foo, y::Foo) = "twofoos"
+ (generic function with 126 methods)

julia> foo1, foo2 = Foo(), Foo()  # Create two Foos
(Foo(), Foo())

julia> +(foo1, foo2)
"twofoos"

julia> foo1 + foo2
"twofoos"
```

We can also create new functions to handle `Foo` data

```
julia> foofunc(x::Foo) = "onefoo"
foofunc (generic function with 1 method)

julia> foofunc(foo)
"onefoo"
```

This example isn't of much use but more useful examples follow

**Composite Data Types** Since the common primitive data types are already built in, most new user-defined data types are composite data types

Composite data types are data types that contain distinct fields of data as attributes

For example, let's say we are doing a lot of work with AR(1) processes, which are random sequences  $\{X_t\}$  that follow a law of motion of the form

$$X_{t+1} = aX_t + b + \sigma W_{t+1} \quad (1.5)$$

Here  $a$ ,  $b$  and  $\sigma$  are scalars and  $\{W_t\}$  is an iid sequence of shocks with some given distribution  $\phi$

At times it might be convenient to take these primitives  $a$ ,  $b$ ,  $\sigma$  and  $\phi$  and organize them into a single entity like so

```
type AR1
    a
    b
    sigma
    phi
end
```

For the distribution  $\phi$  we'll assign a `Distribution` from the `Distributions` package

After reading in the `AR1` definition above we can do the following

```
julia> using Distributions

julia> m = AR1(0.9, 1, 1, Beta(5, 5))
AR1(0.9, 1, 1, Beta( alpha=5.0 beta=5.0 ))
```

In this call to the constructor we've created an instance of `AR1` and bound the name `m` to it

We can access the fields of `m` using their names and "dotted attribute" notation

```
julia> m.a
0.9

julia> m.b
1

julia> m.sigma
1

julia> m.phi
Beta( alpha=5.0 beta=5.0 )
```

For example, the attribute `m.phi` points to an instance of `Beta`, which is in turn a subtype of `Distribution` as defined in the `Distributions` package

```
julia> typeof(m.phi)
Beta (constructor with 3 methods)

julia> typeof(m.phi) <: Distribution
true
```

We can reach in to `m` and change this if we want to

```
julia> m.phi = Exponential(0.5)
Exponential( scale=0.5 )
```

In our type definition we can be explicit that we want phi to be a `Distribution`, and the other elements to be real scalars

```
type AR1
    a::Real
    b::Real
    sigma::Real
    phi)::Distribution
end
```

(Before reading this in you might need to restart your REPL session in order to clear the old definition of `AR1` from memory)

Now the constructor will complain if we try to use the wrong data type

```
julia> m = AR1(0.9, 1, "foo", Beta(5, 5))
ERROR: `convert` has no method matching convert(::Type{Real}, ::ASCIIString) in AR1 at no file
```

This is useful if we're going to have functions that act on instances of `AR1`

- e.g., simulate time series, compute variances, generate histograms, etc.

If those functions only work with `AR1` instances built from the specified data types then it's probably best if we get an error as soon we try to make an instance that doesn't fit the pattern

Better to fail early rather than deeper into our code where errors are harder to debug

**Type Parameters** Consider the following output

```
julia> typeof([10, 20, 30])
Array{Int64,1}
```

Here `Array` is one of Julia's predefined types (`Array <: DenseArray <: AbstractArray <: Any`)

The `Int64,1` in curly brackets are **type parameters**

In this case they are the element type and the dimension

Many other types have type parameters too

```
julia> typeof(1.0 + 1.0im)
Complex{Float64} (constructor with 1 method)

julia> typeof(1 + 1im)
Complex{Int64} (constructor with 1 method)
```

Types with parameters are therefore in fact an indexed family of types, one for each possible value of the parameter

**Defining Parametric Types** We can use parametric types in our own type definitions

Let's say we're defining a type called `FooBar` with attributes `foo` and `bar`

```
type FooBar
    foo
    bar
end
```

Suppose we now decide that we want `foo` and `bar` to have the same type, although we don't much care what that type is

We can achieve this with the syntax

```
type FooBar{T}
    foo::T
    bar::T
end
```

Now our constructor is happy provided that the arguments do in fact have the same type

```
julia> fb = FooBar(1.0, 2.0)
FooBar{Float64}(1.0, 2.0)

julia> fb = FooBar(1, 2)
FooBar{Int64}(1, 2)

julia> fb = FooBar(1, 2.0)
ERROR: `FooBar{T}` has no method matching FooBar{T}(::Int64, ::Float64)
```

Now let's say we want the data to be of the same type *and* that type must be a subtype of `Number`

We can achieve this as follows

```
type FooBar{T <: Number}
    foo::T
    bar::T
end
```

Let's try it

```
julia> fb = FooBar(1, 2)
FooBar{Int64}(1, 2)

julia> fb = FooBar("fee", "fi")
ERROR: `FooBar{T<:Number}` has no method matching FooBar{T<:Number}(::ASCIIString, ::ASCIIString)
```

In the second instance we get an error because `ASCIIString` is not a subtype of `Number`

## Writing Fast Code

Let's briefly discuss how to write Julia code that executes quickly (for a given hardware configuration)

For now our focus is on generating more efficient machine code from essentially the same program (i.e., without parallelization or other more significant changes to the way the program runs)

**Basic Concepts** The benchmark for performance is well written *compiled* code, expressed in languages such as C and Fortran

This is because computer programs are essentially operations on data, and the details of the operations implemented by the CPU depend on the nature of the data

When code is written in a language like C and compiled, the compiler has access to sufficient information to build machine code that will organize the data optimally in memory and implement efficient operations as required for the task in hand

To approach this benchmark, Julia needs to know about the type of data it's processing as early as possible

**An Example** Consider the following function, which essentially does the same job as Julia's `sum()` function but acts only on floating point data

```
function sum_float_array(x::Array{Float64, 1})
    sum = 0.0
    for i in 1:length(x)
        sum += x[i]
    end
    return sum
end
```

Calls to this function run very quickly

```
julia> x_float = linspace(0, 1, int(1e6))
julia> @time sum_float_array(x_float)
elapsed time: 0.002731878 seconds (96 bytes allocated)
```

One reason is that data types are fully specified

When Julia compiles this function via its just-in-time compiler, it knows that the data passed in as `x` will be an array of 64 bit floats

Hence it's known to the compiler that the relevant method for `+` is always addition of floating point numbers

Moreover, the data can be arranged into continuous 64 bit blocks of memory to simplify memory access

Finally, data types are stable — for example, the local variable `sum` starts off as a float and remains a float throughout

**Type Inferences** What happens if we don't supply type information?

Here's the same function minus the type annotation in the function signature

```
function sum_array(x)
    sum = 0.0
    for i in 1:length(x)
        sum += x[i]
    end
```

```
    return sum
end
```

When we run it with the same array of floating point numbers it executes just as fast as before

```
julia> @time sum_array(x_float)
elapsed time: 0.002720878 seconds (96 bytes allocated)
```

The reason is that when `sum_array()` is first called on a vector of a given data type, a newly compiled version of the function is compiled to handle that type

In this case, since we're calling the function on a vector of floats, we get a compiled version of the function with essentially the same internal representation as `sum_float_array()`

Things get tougher for the interpreter when the data type within the array is imprecise

For example, the following snippet creates an array where the element type is `Any`

```
julia> n = int(1e6)
1000000

julia> x_any = {1/i for i in 1:n}

julia> eltype(x_any)
Any
```

Now summation is *much* slower and memory management is less efficient

```
julia> @time sum_array(x_any)
elapsed time: 0.051313847 seconds (16000096 bytes allocated)
```

**Summary and Tips** To write efficient code use functions to segregate operations into logically distinct blocks

Data types will be determined at function boundaries

If types are not supplied then they will be inferred

If types are stable and can be inferred effectively your functions will run fast

**Further Reading** There are many other aspects to writing fast Julia code

A good next stop for further reading is the [relevant part](#) of the Julia documentation

## Exercises

**Exercise 1** Write a function with the signature `simulate(m::AR1, n::Integer, x0::Real)` that takes as arguments

- an instance `m` of `AR1`
- an integer `n`

- a real number  $x_0$

and returns an array containing a time series of length  $n$  generated according to (1.5) where

- the primitives of the AR(1) process are as specified in  $m$
- the initial condition  $X_0$  is set equal to  $x_0$

Here  $AR1$  is as defined above:

```
type AR1
    a::Real
    b::Real
    sigma::Real
    phi::Distribution
end
```

Hint: If  $d$  is an instance of `Distribution` then `rand(d)` generates one random draw from the distribution specified in  $d$

**Exercise 2** The term **universal function** is sometimes applied to functions which

- when called on a scalar return a scalar
- when called on an array of scalars return an array of the same length by acting elementwise on the scalars in the array

For example, `sin()` has this property in Julia

```
julia> sin(pi)
1.2246467991473532e-16

julia> sin([pi, 2pi])
2-element Array{Float64,1}:
 1.22465e-16
 -2.44929e-16
```

Write a universal function  $f$  such that

- $f(k)$  returns a chi-squared random variable with  $k$  degrees of freedom when  $k$  is an integer
- $f(k\_vec)$  returns a vector where  $f(k\_vec)[i]$  is chi-squared with  $k\_vec[i]$  degrees of freedom

Hint: If we take  $k$  independent standard normals, square them all and sum we get a chi-squared with  $k$  degrees of freedom

### Solutions

[Solution notebook](#)

## 1.6 Useful Libraries

## Contents

- *Useful Libraries*
  - *Overview*
  - *Plotting*
  - *Probability*
  - *Working with Data*
  - *Optimization, Roots and Fixed Points*
  - *Others Topics*
  - *Further Reading*

### Overview

While Julia lacks the massive scientific ecosystem of Python, it has successfully attracted a small army of enthusiastic and talented developers

As a result, its package system is moving towards a critical mass of useful, well written libraries

In addition, a major advantage of Julia libraries is that, because Julia itself is sufficiently fast, there is less need to mix in low level languages like C and Fortran

As a result, most Julia libraries are written exclusively in Julia

Not only does this make the libraries more portable, it makes them much easier to dive into, read, learn from and modify

In this lecture we introduce a few of the Julia libraries that we've found particularly useful for quantitative work in economics

### Plotting

There are already several libraries for generating figures in Julia

- Winston
- Gadfly
- PyPlot

Of these, the most mature from the point of view of the end user is PyPlot

In fact PyPlot is just a Julia front end to the excellent Python plotting library Matplotlib

In the following we provide some basic information on how to install and work with this library

**Installing PyPlot** The one disadvantage of PyPlot is that it not only requires Python but also a lot of the scientific Python back end

However, this has become less of a hassle with the advent of the Anaconda Python distribution

Moreover, the scientific Python tools are extremely useful and easily accessible from Julia via [PyCall](#)

We discussed installing Anaconda and PyPlot [here](#)

**Usage** The most important source of information about PyPlot is the [documentation](#) for Matplotlib itself

There are also many useful examples available on the Matplotlib website and elsewhere

**The Procedural API** Matplotlib has a straightforward plotting API that essentially replicates the plotting routines in MATLAB

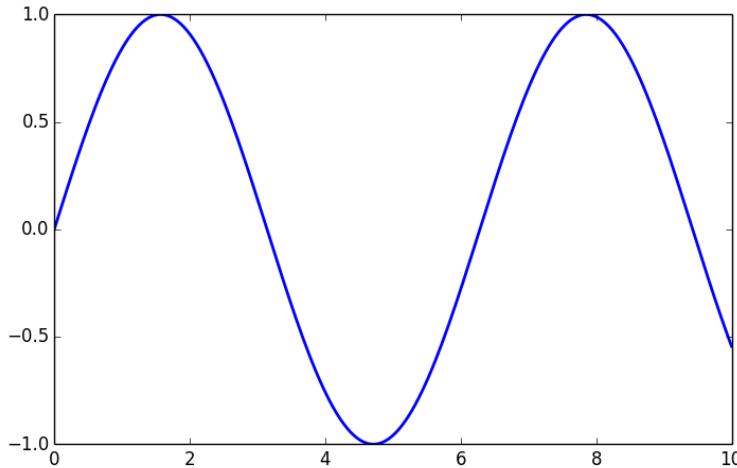
These plotting routines can be expressed in Julia with almost identical syntax

We've already seen some examples of this in earlier lectures

Here's another example

```
using PyPlot
x = linspace(0, 10, 200)
y = sin(x)
plot(x, y, "b-", linewidth=2)
```

The resulting figure looks as follows



**The Object Oriented API** Matplotlib also has a more “Pythonic” object orientated API that power users will prefer

Since Julia doesn't bundle objects with methods in the same way that Python does, plots based on this API don't follow exactly the same syntax that they do in Matplotlib

Fortunately the differences are consistent and after seeing some examples you will find it easy to translate one into the other

Here's an example of the syntax we're discussing, which in this case generates exactly the same plot

```
using PyPlot
x = linspace(0, 10, 200)
y = sin(x)
fig, ax = subplots()
ax[:plot](x, y, "b-", linewidth=2)
```

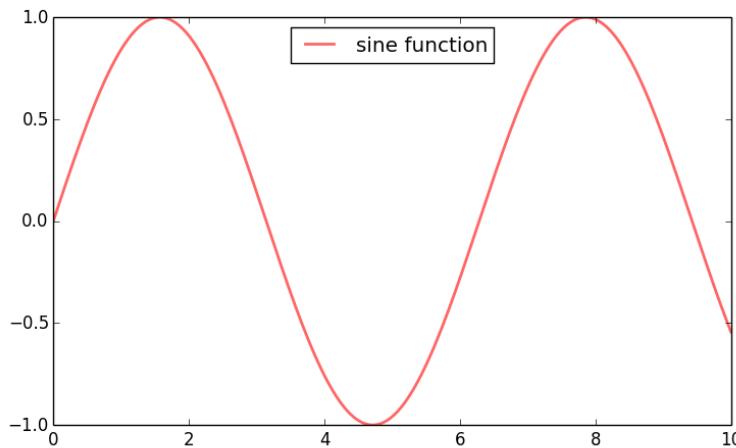
In this case we get nothing extra and have to accept more complexity and a less attractive syntax

However, it is a little more explicit and this turns out to be convenient as we move to more sophisticated plots

Here's a similar plot with a bit more customization

```
using PyPlot
x = linspace(0, 10, 200)
y = sin(x)
fig, ax = subplots()
ax[:plot](x, y, "r-", linewidth=2, label="sine function", alpha=0.6)
ax[:legend](loc="upper center")
```

The resulting figure has a legend at the top center



We can render the legend in LaTeX by changing the `ax[:plot]` line to

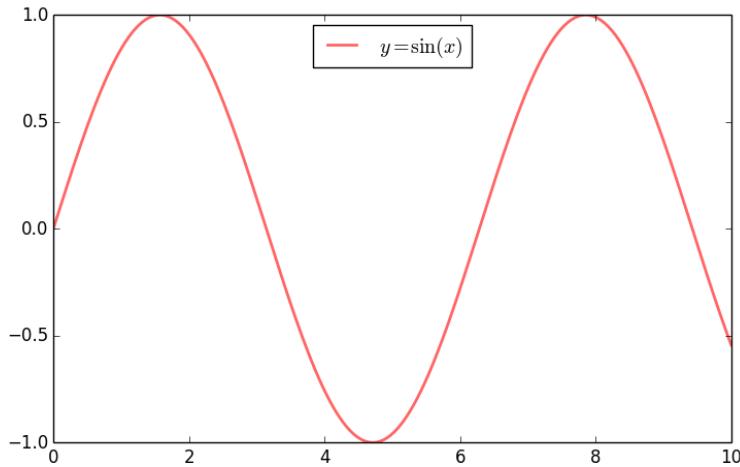
```
ax[:plot](x, y, "r-", linewidth=2, label=L"\$y = \sin(x)\$", alpha=0.6)
```

Note the L in front of the string to indicate LaTeX mark up

The result looks as follows

Here's another example, which helps illustrate how to put multiple plots on one figure

```
using PyPlot
using Distributions
```



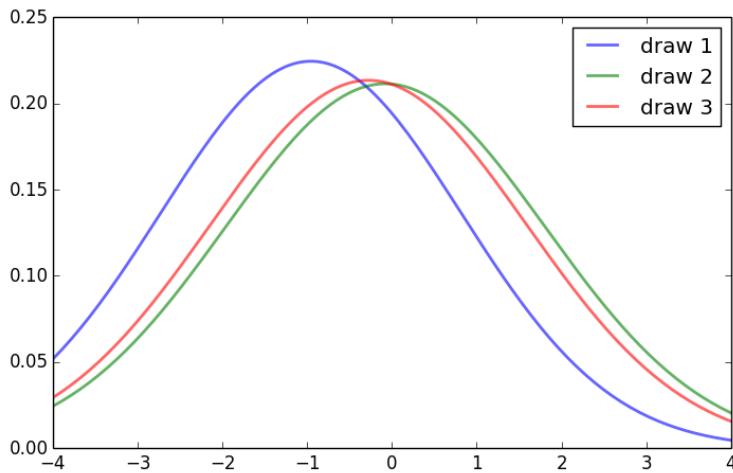
```

u = Uniform()

fig, ax = subplots()
x = linspace(-4, 4, 150)
for i in 1:3
    # == Compute normal pdf from randomly generated mean and std ==
    m, s = rand(u) * 2 - 1, rand(u) + 1
    d = Normal(m, s)
    y = pdf(d, x)
    # == Plot current pdf ==
    ax[:plot](x, y, linewidth=2, alpha=0.6, label="draw $i")
end
ax[:legend]()

```

It generates the following plot



**Multiple Subplots** A figure containing  $n$  rows and  $m$  columns of subplots can be created by the call

```
fig, axes = subplots(num_rows, num_cols)
```

Here's an example

```
using PyPlot
using Distributions

u = Uniform()
num_rows, num_cols = 3, 2
fig, axes = plt.subplots(num_rows, num_cols, figsize=(8, 12))
subplot_num = 0

for i in 1:num_rows
    for j in 1:num_cols
        ax = axes[i, j]
        subplot_num += 1
        # == Generate a normal sample with random mean and std ==
        m, s = rand(u) * 2 - 1, rand(u) + 1
        d = Normal(m, s)
        x = rand(d, 100)
        # == Histogram the sample ==
        ax[:hist](x, alpha=0.6, bins=20)
        ax[:set_title]("histogram $subplot_num")
        ax[:set_xticks]([-4, 0, 4])
        ax[:set_yticks]()
    end
end
```

The resulting figure is as follows

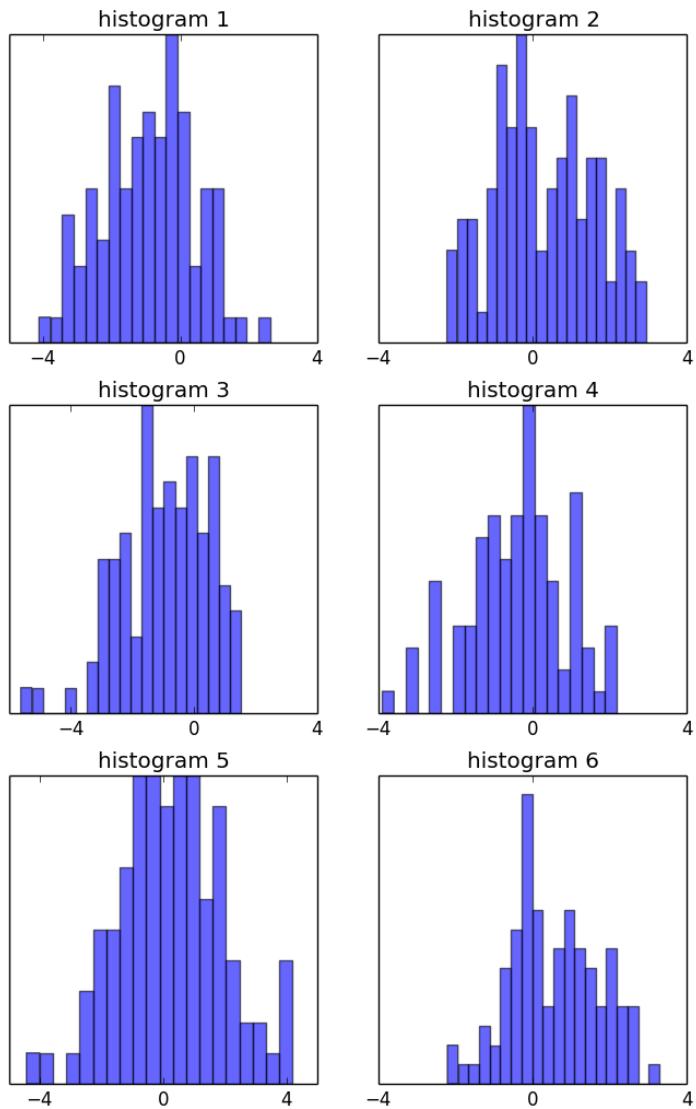
**3D Plots** Here's an example of how to create a 3D plot

```
using PyPlot
using Distributions
using QuantEcon: meshgrid

n = 50
x = linspace(-3, 3, n)
y = x

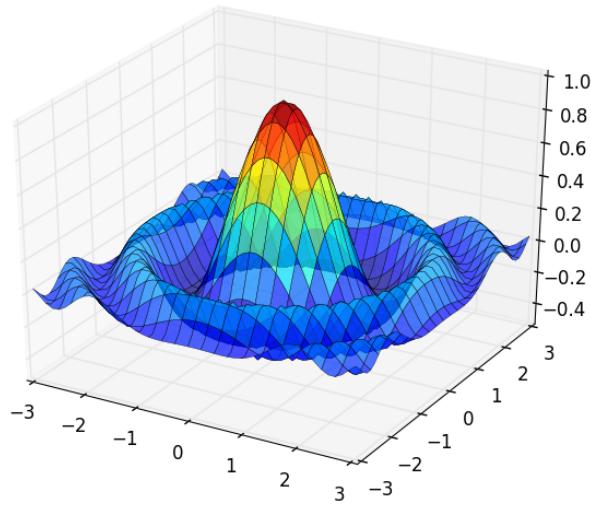
z = Array(Float64, n, n)
f(x, y) = cos(x^2 + y^2) / (1 + x^2 + y^2)
for i in 1:n
    for j in 1:n
        z[j, i] = f(x[i], y[j])
    end
end

fig = figure(figsize=(8,6))
ax = fig[:gca](projection="3d")
```



```
ax[:set_zlim](-0.5, 1.0)
xgrid, ygrid = meshgrid(x, y)
ax[:plot_surface](xgrid, ygrid, z, rstride=2, cstride=2, cmap=ColorMap("jet"), alpha=0.7, linewidth=0.25)
```

It creates this figure



## Probability

Functions for manipulating probability distributions and generating random variables are supplied by the excellent [Distributions](#) package

This package has [first-rate documentation](#) so we'll restrict ourselves to a few comments

The calls to create instances of various random variables take the form `d = DistributionName(params)`

Here are several examples

- Normal with mean  $m$  and standard deviation  $s$ 
  - `d = Normal(m, s)` with defaults  $m = 0$  and  $s = 1$
- Uniform on interval  $[a, b]$ 
  - `d = Uniform(a, b)` with defaults  $a = 0$  and  $b = 1$
- Binomial over  $n$  trials with success probability  $p$ 
  - `d = Binomial(n, p)` with defaults  $n = 1$  and  $p = 0.5$

The Distributions package defines many methods for acting on these instances in order to obtain

- random draws
- pdf (density), cdf (distribution function), quantiles, etc.
- mean, variance, kurtosis, etc.

For example,

- To generate  $k$  draws from the instance  $d$  use `rand(d, k)`
- To obtain the mean of the distribution use `mean(d)`
- To evaluate the probability density function of  $d$  at  $x$  use `pdf(d, x)`

Further details on the interface can be found [here](#)

Several multivariate distributions are also implemented

### Working with Data

A useful package for working with data is `DataFrames`

The most important data type provided is a `DataFrame`, a two dimensional array for storing heterogeneous data

Although data can be heterogeneous within a `DataFrame`, the contents of the columns must be homogeneous

This is analogous to a `data.frame` in R, a `DataFrame` in Pandas (Python) or, more loosely, a spreadsheet in Excel

The `DataFrames` package also supplies a `DataArray` type, which is like a one dimensional `DataFrame`

In terms of working with data, the advantage of a `DataArray` over a standard numerical array is that it can handle missing values

Here's an example

```
julia> using DataFrames

julia> commodities = ["crude", "gas", "gold", "silver"]
4-element Array{ASCIIString,1}:
 "crude"
 "gas"
 "gold"
 "silver"

julia> last = @data([4.2, 11.3, 12.1, NA]) # Create DataArray
4-element DataArray{Float64,1}:
 4.2
 11.3
 12.1
 NA

julia> df = DataFrame(commod = commodities, price = last)
```

4x2 DataFrame		
Row #	commod	price
1	"crude"	4.2
2	"gas"	11.3
3	"gold"	12.1
4	"silver"	NA

Columns of the DataFrame can be accessed by name

```
julia> df[:price]
4-element DataArray{Float64,1}:
 4.2
 11.3
 12.1
 NA

julia> df[:commod]
4-element DataArray{ASCIIString,1}:
 "crude"
 "gas"
 "gold"
 "silver"
```

The `DataFrames` package provides a number of methods for acting on DataFrames

A simple one is `describe()`

```
julia> describe(df)
commod
Length 4
Type ASCIIString
NAs 0
NA% 0.0%
Unique 4

price
Min 4.2
1st Qu. 7.75
Median 11.3
Mean 9.200000000000001
3rd Qu. 11.7
Max 12.1
NAs 1
NA% 25.0%
```

There are also functions for splitting, merging and other data munging operations

Data can be read from and written to CSV files using syntax `df = readtable("data_file.csv")` and `writetable("data_file.csv", df)` respectively

Other packages for working with data can be found at [JuliaStats](#) and [JuliaQuant](#)

### Optimization, Roots and Fixed Points

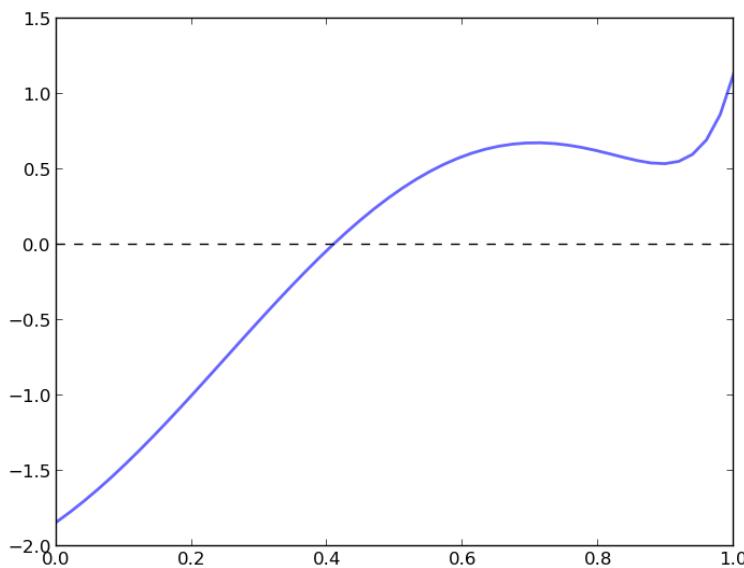
Let's look briefly at the optimization and root finding algorithms

**Roots** 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

$$f(x) = \sin(4(x - 1/4)) + x + x^{20} - 1 \quad (1.6)$$

with  $x \in [0, 1]$  we get



The unique root is approximately 0.408

One common root-finding algorithm is the [Newton-Raphson method](#)

This is implemented as `newton()` in the [Roots](#) package and is called with the function and an initial guess

```
julia> using Roots

julia> f(x) = sin(4 * (x - 1/4)) + x + x^20 - 1
f (generic function with 1 method)

julia> newton(f, 0.2)
0.40829350427936706
```

The Newton-Raphson method uses local slope information, which can lead to failure of convergence for some initial conditions

```
julia> newton(f, 0.7)
-1.0022469256696989
```

For this reason most modern solvers use more robust “hybrid methods”, as does Roots’s `fzero()` function

```
julia> fzero(f, 0, 1)
0.40829350427936706
```

**Optimization** For constrained, univariate minimization a useful option is `optimize()` from the `Optim` package

This function defaults to a robust hybrid optimization routine called Brent’s method

```
julia> using Optim

julia> optimize(x -> x^2, -1.0, 1.0)
Results of Optimization Algorithm
* Algorithm: Brent's Method
* Search Interval: [-1.000000, 1.000000]
* Minimum: -0.000000
* Value of Function at Minimum: 0.000000
* Iterations: 5
* Convergence: max(|x - x_upper|, |x - x_lower|) <= 2*(1.5e-08|x|+2.2e-16): true
* Objective Function Calls: 6
```

For other optimization routines, including least squares and multivariate optimization, see [the documentation](#)

A number of alternative packages for optimization can be found at [JuliaOpt](#)

## Others Topics

**Numerical Integration** The base library contains a function called `quadgk()` that performs Gaussian quadrature

```
julia> quadgk(x -> cos(x), -2pi, 2pi)
(5.644749237155177e-15, 4.696156369056425e-22)
```

This is an adaptive Gauss-Kronrod integration technique that’s relatively accurate for smooth functions

However, its adaptive implementation makes it slow and not well suited to inner loops

For this kind of integration you can use the quadrature routines from QuantEcon

```
julia> using QuantEcon

julia> nodes, weights = qnwlege(65, -2pi, 2pi);

julia> integral = do_quad(x -> cos(x), nodes, weights)
-2.912600716165059e-15
```

Let's time the two implementations

```
julia> @time quadgk(x -> cos(x), -2pi, 2pi)
elapsed time: 2.732162971 seconds (984420160 bytes allocated, 40.55% gc time)

julia> @time do_quad(x -> cos(x), nodes, weights)
elapsed time: 0.002805691 seconds (1424 bytes allocated)
```

We get similar accuracy with a speed up factor approaching three orders of magnitude

More numerical integration (and differentiation) routines can be found in the package [Calculus](#)

**Linear Algebra** The standard library contains many useful routines for linear algebra, in addition to standard functions such as `det()`, `inv()`, `eye()`, etc.

Routines are available for

- Cholesky factorization
- LU decomposition
- Singular value decomposition,
- Schur factorization, etc.

See [here](#) for further details

### Further Reading

The full set of libraries available under the Julia packaging system can be browsed at [pkg.julialang.org](http://pkg.julialang.org)



---

CHAPTER  
TWO

---

## INTRODUCTORY APPLICATIONS

This section of the course contains intermediate and foundational applications.

### 2.1 Linear Algebra

#### Contents

- *Linear Algebra*
  - *Overview*
  - *Vectors*
  - *Matrices*
  - *Solving Systems of Equations*
  - *Eigenvalues and Eigenvectors*
  - *Further Topics*

#### Overview

One of the single most useful branches of mathematics you can learn is linear algebra

For example, many applied problems in economics, finance, operations research and other fields of science 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 Julia 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

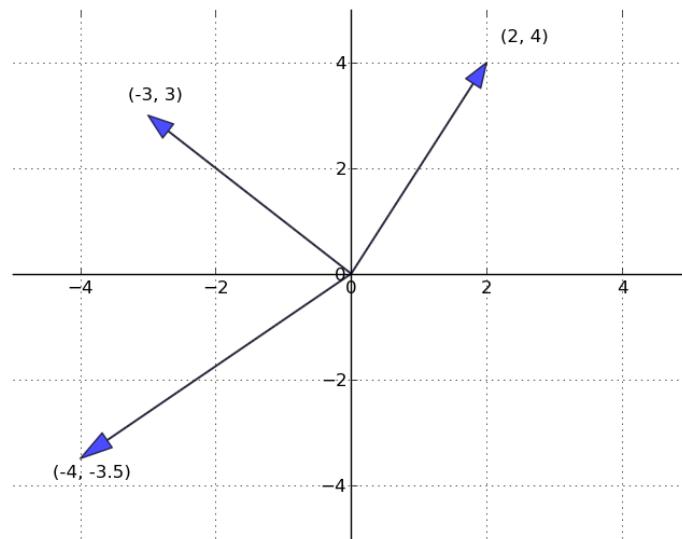
(Later, when we wish to perform certain matrix operations, it will become necessary to distinguish between the two)

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 Julia 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

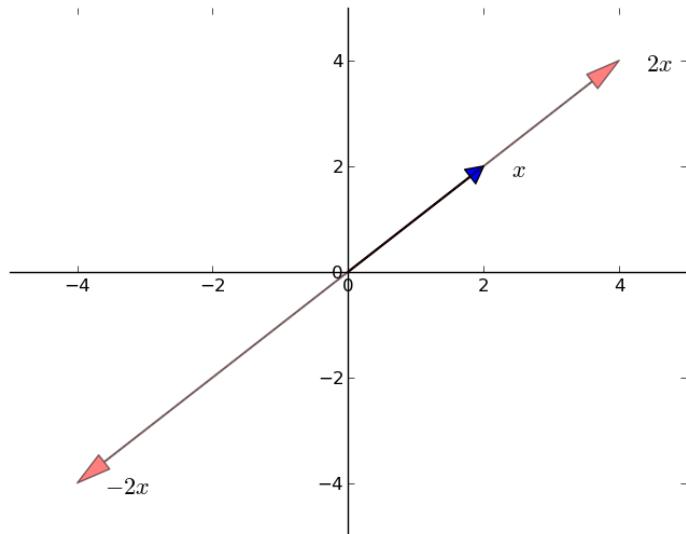
As a matter of definition, when we add two vectors, we add them element by element

$$x + y = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} := \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix}$$

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 Julia, a vector can be represented as a one dimensional *Array*

Julia *Arrays* allow us to express scalar multiplication and addition with a very natural syntax

```
julia> x = ones(3)
3-element Array{Float64,1}:
 1.0
 1.0
 1.0
```

```
julia> y = [2, 4, 6]
3-element Array{Int64,1}:
 2
 4
 6

julia> x + y
3-element Array{Float64,1}:
 3.0
 5.0
 7.0

julia> 4x # equivalent to 4 * x and 4 .* x
3-element Array{Float64,1}:
 4.0
 4.0
 4.0
```

**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

```
julia> dot(x, y)          # Inner product of x and y
12.0

julia> sum(x .* y)       # Gives the same result
12.0

julia> norm(x)           # Norm of x
1.7320508075688772

julia> sqrt(sum(x.^2))   # Gives the same result
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

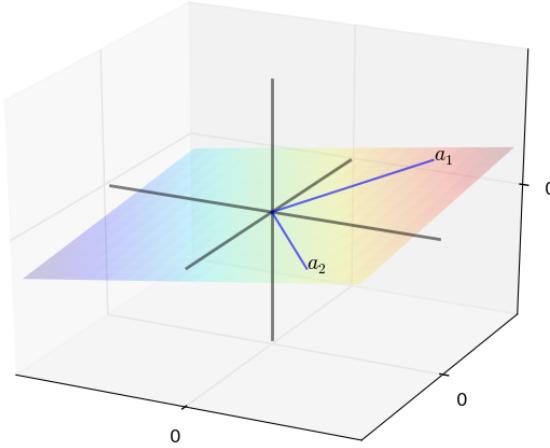
$$y = \beta_1 a_1 + \cdots + \beta_k a_k \text{ for some scalars } \beta_1, \dots, \beta_k$$

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:

$$\gamma A = \gamma \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} := \begin{bmatrix} \gamma a_{11} & \cdots & \gamma a_{1k} \\ \vdots & \vdots & \vdots \\ \gamma a_{n1} & \cdots & \gamma a_{nk} \end{bmatrix}$$

and

$$A + B = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} + \begin{bmatrix} b_{11} & \cdots & b_{1k} \\ \vdots & \vdots & \vdots \\ b_{n1} & \cdots & b_{nk} \end{bmatrix} := \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1k} + b_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} + b_{n1} & \cdots & a_{nk} + b_{nk} \end{bmatrix}$$

In the latter case, the matrices must have the same shape in order for the definition to make sense

We also have a convention for *multiplying* two matrices

The rule for matrix multiplication generalizes the idea of inner products discussed above, and is designed to make multiplication play well with basic linear operations

If  $A$  and  $B$  are two matrices, then their product  $AB$  is formed by taking as its  $i, j$ -th element the inner product of the  $i$ -th row of  $A$  and the  $j$ -th column of  $B$

There are many tutorials to help you visualize this operation, such as [this one](#), or the discussion on the [Wikipedia page](#)

If  $A$  is  $n \times k$  and  $B$  is  $j \times m$ , then to multiply  $A$  and  $B$  we require  $k = j$ , and the resulting matrix  $AB$  is  $n \times m$

As perhaps the most important special case, consider multiplying  $n \times k$  matrix  $A$  and  $k \times 1$  column vector  $x$

According to the preceding rule, this gives us an  $n \times 1$  column vector

$$Ax = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} := \begin{bmatrix} a_{11}x_1 + \cdots + a_{1k}x_k \\ \vdots \\ a_{n1}x_1 + \cdots + a_{nk}x_k \end{bmatrix} \quad (2.2)$$

**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 Julia** Julia arrays are also used as matrices, and have fast, efficient functions and methods for all the standard matrix operations

You can create them as follows

```
julia> A = [1 2
           3 4]
2x2 Array{Int64,2}:
 1  2
 3  4

julia> typeof(A)
Array{Int64,2}

julia> size(A)
(2,2)
```

The `size` function returns a tuple giving the number of rows and columns

To get the transpose of  $A$ , use `transpose(A)` or, more simply,  $A'$

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

```
julia> A = eye(3)
3x3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

```

0.0  0.0  1.0

julia> B = ones(3, 3)
3x3 Array{Float64,2}:
 1.0  1.0  1.0
 1.0  1.0  1.0
 1.0  1.0  1.0

julia> 2A
3x3 Array{Float64,2}:
 2.0  0.0  0.0
 0.0  2.0  0.0
 0.0  0.0  2.0

julia> A + B
3x3 Array{Float64,2}:
 2.0  1.0  1.0
 1.0  2.0  1.0
 1.0  1.0  2.0

```

To multiply matrices we use the `*` operator

In particular, `A * B` is matrix multiplication, whereas `A .* B` is element by element multiplication

**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

$$y = Ax \tag{2.3}$$

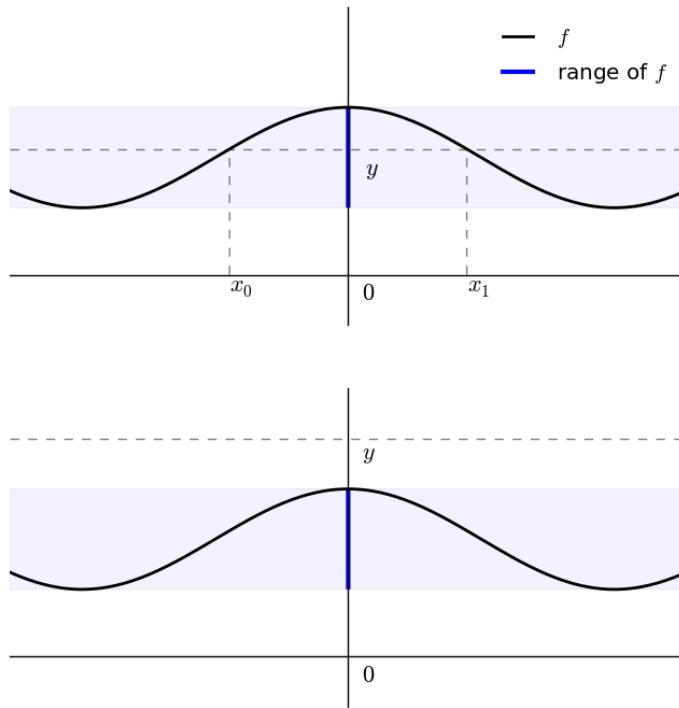
The problem we face is to determine a vector  $x \in \mathbb{R}^k$  that solves (2.3), taking  $y$  and  $A$  as given

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

$$Ax = x_1 a_1 + \cdots + x_k a_k$$

Hence the range of  $f(x) = Ax$  is exactly the span of the columns of  $A$

We want the range to be large, so that it contains arbitrary  $y$

As you might recall, the condition that we want for the span to be large is *linear independence*

A happy fact is that linear independence of the columns of  $A$  also gives us uniqueness

Indeed, it follows from our *earlier discussion* that if  $\{a_1, \dots, a_k\}$  are linearly independent and  $y = Ax = x_1 a_1 + \cdots + x_k a_k$ , then no  $z \neq x$  satisfies  $y = Az$

**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 2 vectors are enough to span  $\mathbb{R}^2$

(For example, use the canonical basis vectors)

It follows that one column is a linear combination of the other two

For example, let's say that  $a_1 = \alpha a_2 + \beta a_3$

Then if  $y = Ax = x_1 a_1 + x_2 a_2 + x_3 a_3$ , we can also write

$$y = x_1(\alpha a_2 + \beta a_3) + x_2 a_2 + x_3 a_3 = (x_1\alpha + x_2)a_2 + (x_1\beta + x_3)a_3$$

In other words, uniqueness fails

**Linear Equations with Julia** Here's an illustration of how to solve linear equations with Julia's built-in linear algebra facilities

```
julia> A = [1.0 2.0; 3.0 4.0];
julia> y = ones(2, 1); # A column vector
```

```
julia> det(A)
-2.0

julia> A_inv = inv(A)
2x2 Array{Float64,2}:
 -2.0   1.0
  1.5  -0.5

julia> x = A_inv * y  # solution
2x1 Array{Float64,2}:
 -1.0
  1.0

julia> A * x  # should equal y (a vector of ones)
2x1 Array{Float64,2}:
 1.0
 1.0

julia> A\y  # produces the same solution
2x1 Array{Float64,2}:
 -1.0
  1.0
```

Observe how we can solve for  $x = A^{-1}y$  by either via `inv(A) * y`, or using `A \ y`

The latter method is preferred because it automatically selects the best algorithm for the problem based on the values of  $A$  and  $y$

If  $A$  is not square then  $A \ y$  returns the least squares solution  $\hat{x} = (A'A)^{-1}A'y$

### 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)

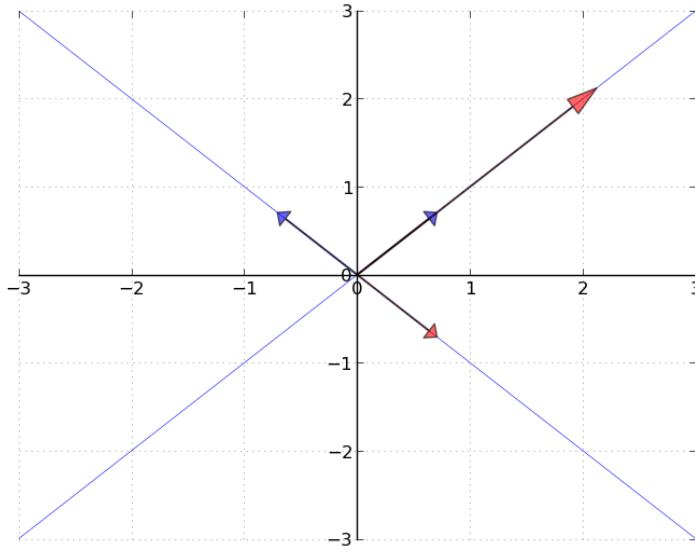
As expected, the image  $Av$  of each  $v$  is just a scaled version of the original

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 Julia, we can solve for the eigenvalues and eigenvectors of a matrix as follows

```
julia> A = [1.0 2.0; 2.0 1.0];
julia> evals, evecs = eig(A);
julia> evals
2-element Array{Float64,1}:
 -1.0
  3.0
julia> evecs
2x2 Array{Float64,2}:
 -0.707107  0.707107
  0.707107  0.707107
```

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 Julia via `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>1</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

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k \tag{2.4}$$

---

<sup>1</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.

**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
- $B$  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  $v$  satisfies  $v(x) = -x'\tilde{P}x$  where  $\tilde{P} = A'PA - A'PB(Q + B'PB)^{-1}B'PA$

As we will see, in economic contexts Lagrange multipliers often are shadow prices

**Note:** If we don't care about the Lagrange multipliers, we can substitute the constraint into the objective function, and then just maximize  $-(Ax + Bu)'P(Ax + Bu) - u'Qu$  with respect to  $u$ . You can verify that this leads to the same maximizer.

**Further Reading** The documentation of the linear algebra features built into Julia 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 [\[Janich94\]](#)

## 2.2 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

Attributes:

- simple, flexible and supported by many elegant theoretical results
- valuable for building intuition about random dynamic models
- very useful 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 great 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  $P = P[s, s']$  such that

1. each element  $P[s, s']$  is nonnegative, and
2. each row  $P[s, \cdot]$  sums to one

(The square brackets notation for indices is unconventional but ties in well with the code below and helps us differentiate between indices and time subscripts)

Let  $S := \{1, \dots, n\}$

Each row  $P[s, \cdot]$  can be regarded as a distribution (probability mass function) on  $S$

It is not difficult to check <sup>2</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

A *Markov chain*  $\{X_t\}$  on  $S$  is a stochastic process on  $S$  that has the *Markov property*

This means that, for any date  $t$  and any state  $s' \in S$ ,

$$\mathbb{P}\{X_{t+1} = s' | X_t\} = \mathbb{P}\{X_{t+1} = s' | X_t, X_{t-1}, \dots\} \quad (2.5)$$

In other words, knowing the current state is enough to understand probabilities for future states

In particular, the dynamics of a Markov chain are fully determined by the set of values

$$P[s, s'] := \mathbb{P}\{X_{t+1} = s' | X_t = s\} \quad (s, s' \in S)$$

By construction,

- $P[s, s']$  is the probability of going from  $s$  to  $s'$  in one unit of time (one step)
- $P[s, \cdot]$  is the conditional distribution of  $X_{t+1}$  given  $X_t = s$

It's clear that  $P$  is a stochastic matrix

Conversely, 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  $t = 0, 1, \dots$ ,
  - draw  $X_{t+1}$  from  $P[X_t, \cdot]$

By construction, the resulting process satisfies (2.5)

**Example 1** Consider a worker who, at any given time  $t$ , is either unemployed (state 1) or employed (state 2)

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 stochastic matrix, this tells us that  $P[1, 2] = \alpha$  and  $P[2, 1] = \beta$ , or

$$P = \begin{pmatrix} 1 - \alpha & \alpha \\ \beta & 1 - \beta \end{pmatrix}$$

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?

---

<sup>2</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.

- Conditional on employment, what is the probability of becoming unemployed at least once over the next 12 months?
- Etc.

We'll cover such applications below

**Example 2** Using US unemployment data, Hamilton [Ham05] estimated the stochastic matrix

$$P := \begin{pmatrix} 0.971 & 0.029 & 0 \\ 0.145 & 0.778 & 0.077 \\ 0 & 0.508 & 0.492 \end{pmatrix}$$

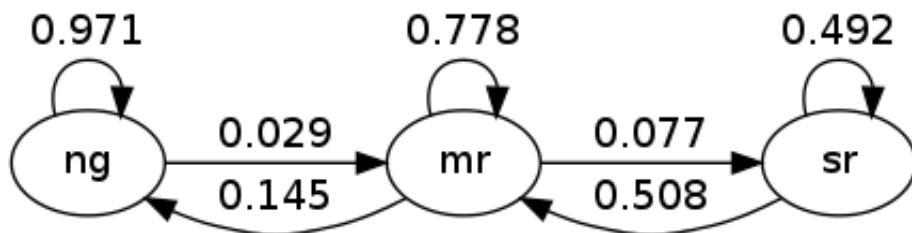
where

- 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 of the most natural ways to answer questions about Markov chains is to simulate them

(As usual, 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

This is probably what you should use in applications, since it's efficient and 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

**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

```
julia> using QuantEcon

julia> psi = [0.1, 0.9];      # Probabilities over sample space {1, 2}

julia> d = DiscreteRV(psi);

julia> draw(d, 5)           # Generate 5 independent draws from psi
5-element Array{Int64,1}:
 1
 2
 2
 1
 2
```

We'll write our code as a function that takes the following three arguments

- A stochastic matrix  $P$
- An initial state `init`
- A positive integer `sample_size` representing the length of the time series the function should return

```
using QuantEcon

function mc_sample_path(P; init=1, sample_size=1000)
    X = Array(Int64, sample_size) # allocate memory
    X[1] = init
    # === convert each row of P into a distribution === #
    n = size(P)[1]
    P_dist = [DiscreteRV(vec(P[i,:])) for i in 1:n]

    # === generate the sample path === #
    for t in 1:(sample_size - 1)
        X[t+1] = draw(P_dist[X[t]])
    end
    return X
end
```

Let's see how it works using the small matrix

$$P := \begin{pmatrix} 0.4 & 0.6 \\ 0.2 & 0.8 \end{pmatrix} \quad (2.6)$$

As we'll see later, for a long series drawn from  $P$ , the fraction of the sample that takes value 1 will be about 0.25

If you run the following code you should get roughly that answer

```
julia> P = [0.4 0.6; 0.2 0.8]
2x2 Array{Float64,2}:
 0.4  0.6
 0.2  0.8

julia> X = mc_sample_path(P, sample_size=100000);

julia> println(mean(X .== 1))
0.25171
```

**Using QuantEcon's Routines** As discussed above, QuantEcon has very nice routines for handling Markov chains, including simulation

Here's an illustration using the same  $P$  as the preceding example

```
julia> using QuantEcon

julia> mc = MarkovChain(P)
Discrete Markov Chain
stochastic matrix:
2x2 Array{Float64,2}:
 0.4  0.6
 0.2  0.8

julia> P = [.4 .6; .2 .8];

julia> X = mc_sample_path(mc, [0.5, 0.5], 100000);

julia> println(mean(X .== 1))  # Should be about 0.25
0.24883
```

### 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  $X_{t+m}$ ?

(Motivation for these questions is given below)

**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  $s' \in S$

Using the [law of total probability](#), we can decompose the probability that  $X_{t+1} = s'$  as follows:

$$\mathbb{P}\{X_{t+1} = s'\} = \sum_{s \in S} \mathbb{P}\{X_{t+1} = s' | X_t = s\} \cdot \mathbb{P}\{X_t = s\}$$

In words, to get the probability of being at  $s'$  tomorrow, we account for all ways this can happen and sum their probabilities

Rewriting this statement in terms of marginal and conditional probabilities gives

$$\psi_{t+1}[s'] = \sum_{s \in S} P[s, s'] \psi_t[s]$$

There are  $n$  such equations, one for each  $s' \in S$

If we think of  $\psi_{t+1}$  and  $\psi_t$  as *row vectors* (as is traditional), these  $n$  equations are summarized by the matrix expression

$$\psi_{t+1} = \psi_t P \tag{2.7}$$

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 (2.7), 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

$$X_0 \sim \psi_0 \implies X_m \sim \psi_0 P^m \tag{2.8}$$

and, more generally,

$$X_t \sim \psi_t \implies X_{t+m} \sim \psi_t P^m \tag{2.9}$$

**Multiple Step Transition Probabilities** We know that the probability of transitioning from  $s$  to  $s'$  in one step is  $P[s, s']$

It turns out that the probability of transitioning from  $s$  to  $s'$  in  $m$  steps is  $P^m[s, s']$ , the  $[s, s']$ -th element of the  $m$ -th power of  $P$

To see why, consider again (2.9), but now with  $\psi_t$  putting all probability on state  $s$

If we regard  $\psi_t$  as a vector, it is a vector with 1 in the  $s$ -th position and zero elsewhere

Inserting this into (2.9), we see that, conditional on  $X_t = s$ , the distribution of  $X_{t+m}$  is the  $s$ -th row of  $P^m$

In particular

$$\mathbb{P}\{X_{t+m} = s'\} = P^m[s, s'] = [s, s']\text{-th element of } P^m$$

**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  $s$  to be  $\psi[s]$

The probability of being in recession (state 1 or state 2) 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** 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  $\{1, 2\}$

- For example,  $\psi[1]$  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

Just about every theoretical treatment of Markov chains has some discussion of the concepts of irreducibility and aperiodicity

Let's see what they're about

**Irreducibility** Let  $P$  be a fixed stochastic matrix

Two states  $s$  and  $s'$  are said to *communicate* with each other if there exist positive integers  $j$  and  $k$  such that

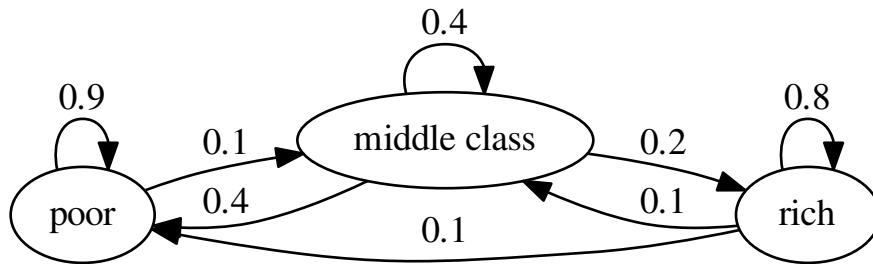
$$P^j[s, s'] > 0 \quad \text{and} \quad P^k[s', s] > 0$$

In view of our discussion *above*, this means precisely that

- state  $s$  can be reached eventually from state  $s'$ , and
- state  $s'$  can be reached eventually from state  $s$

The stochastic matrix  $P$  is called *irreducible* if all states communicate; that is, if  $s$  and  $s'$  communicate for all  $s, s'$  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

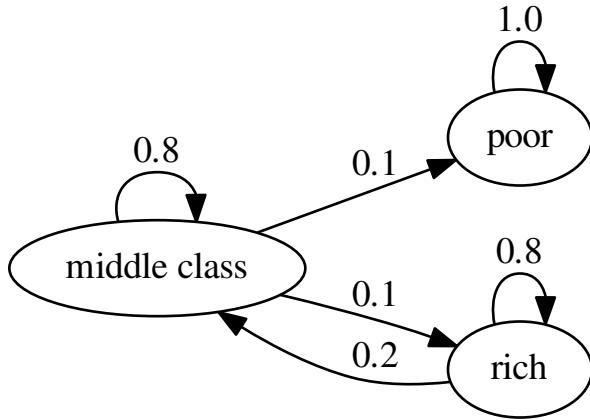
```
julia> using QuantEcon

julia> P = [0.9 0.1 0.0; 0.4 0.4 0.2; 0.1 0.1 0.8];

julia> mc = MarkovChain(P)
Discrete Markov Chain
stochastic matrix:
3x3 Array{Float64,2}:
 0.9  0.1  0.0
 0.4  0.4  0.2
 0.8  0.1  0.1

julia> is_irreducible(mc)
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

```

julia> using QuantEcon
julia> P = [1.0 0.0 0.0; 0.1 0.8 0.1; 0.0 0.2 0.8];
julia> mc = MarkovChain(P);
julia> is_irreducible(mc)
false
  
```

We can also determine the “communication classes”

```

julia> communication_classes(mc)
2-element Array{Array{Int64,1},1}:
 [1]
 [2,3]
  
```

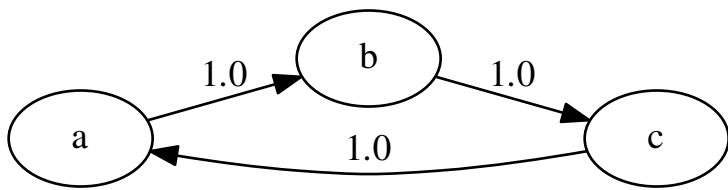
It might be clear to you already that irreducibility is going to be important in terms of long run outcomes

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:

```

julia> using QuantEcon
julia> P = [0 1 0; 0 0 1; 1 0 0];
julia> mc = MarkovChain(P);
julia> period(mc)
3

```

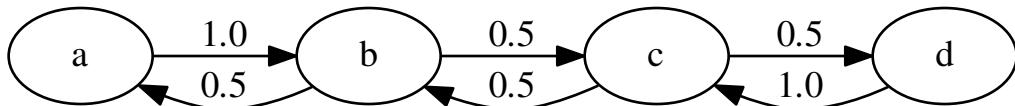
More formally, the *period* of a state  $s$  is the greatest common divisor of the set of integers

$$D(s) := \{j \geq 1 : P^j[s, s] > 0\}$$

In the last example,  $D(s) = \{3, 6, 9, \dots\}$  for every state  $s$ , 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

```

julia> P = zeros(4, 4);
julia> P[1,2] = 1;
julia> P[2, 1] = P[2, 3] = 0.5;
julia> P[3, 2] = P[3, 4] = 0.5;

```

```
julia> P[4, 3] = 1;
julia> mc = MarkovChain(P);
julia> period(mc)
2
julia> is_aperiodic(mc)
false
```

## Stationary Distributions

As seen in (2.7), we can shift probabilities forward one unit of time via postmultiplication by  $P$ . Some distributions are invariant under this updating process — for example,

```
julia> P = [.4 .6; .2 .8];
julia> psi = [0.25, 0.75];
julia> psi' * P
1x2 Array{Float64,2}:
 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 just 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)

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 1)

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

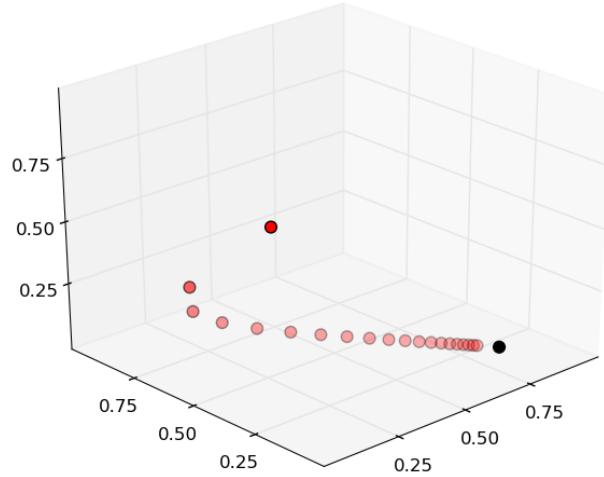
```
using QuantEcon P = [.4 .6; .2 .8] mc = MarkovChain(P)
println(mc_compute_stationary(mc))
```

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 library — you might like to try experimenting with different initial conditions

### Ergodicity

Under irreducibility, yet another important result obtains: For all  $s \in S$ ,

$$\frac{1}{n} \sum_{t=1}^n \mathbf{1}\{X_t = s\} \rightarrow \psi^*[s] \quad \text{as } n \rightarrow \infty \quad (2.10)$$

Here

- $\mathbf{1}\{X_t = s\} = 1$  if  $X_t = s$  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  $s$  converges to  $\psi^*[s]$  as time goes to infinity. This gives us another way to interpret the stationary distribution — provided that the convergence result in (2.10) is valid.

The convergence in (2.10) is a special case of a law of large numbers result for Markov chains — see [EDTC](#), section 4.3.4 for some additional information.

**Example** Recall our cross-sectional interpretation of the employment / unemployment model *discussed above*

Assume that  $\alpha \in (0, 1)$  and  $\beta \in (0, 1)$ , so that irreducibility and aperiodicity both hold.

We saw that the stationary distribution is  $(p, 1 - p)$ , where

$$p = \frac{\beta}{\alpha + \beta}$$

In the cross-sectional interpretation, this is the fraction of people unemployed.

In view of our latest (ergodicity) result, it is also the fraction of time that a worker can expect to spend unemployed.

Thus, in the long-run, cross-sectional averages for a population and time-series averages for a given person coincide.

This is one interpretation of the notion of ergodicity.

### Computing Expectations

We are interested in computing expectations of the form

$$\mathbb{E}[h(X_t)] \tag{2.11}$$

and conditional expectations such as

$$\mathbb{E}[h(X_{t+k}) \mid X_t = s] \tag{2.12}$$

where

- $\{X_t\}$  is a Markov chain generated by  $n \times n$  stochastic matrix  $P$
- $h: S \rightarrow \mathbb{R}$  is a given function

The unconditional expectation (2.11) is easy: We just need to sum over the distribution of  $X_t$ .

That is, we take

$$\mathbb{E}[h(X_t)] = \sum_{s \in S} (\psi P^t)[s] h[s]$$

where  $\psi$  is the distribution of  $X_0$ .

In this setting it's traditional to regard  $h$  as a column vector, in which case the expression can be rewritten more simply as

$$\mathbb{E}[h(X_t)] = \psi P^t h$$

As for the conditional expectation (2.12), we need to sum over the conditional distribution of  $X_{t+k}$  given  $X_t = s$

We already know that this is  $P^k[s, \cdot]$ , so

$$\mathbb{E}[h(X_{t+k}) \mid X_t = s] = (P^k h)[s] \quad (2.13)$$

The vector  $P^k h$  stores the conditional expectation  $\mathbb{E}[h(X_{t+k}) \mid X_t = s]$  over all  $s$

**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

$$\mathbb{E} \left[ \sum_{j=0}^{\infty} \beta^j h(X_{t+j}) \mid X_t = s \right] = [(I - \beta P)^{-1} h][s]$$

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

$$p := \frac{\beta}{\alpha + \beta}$$

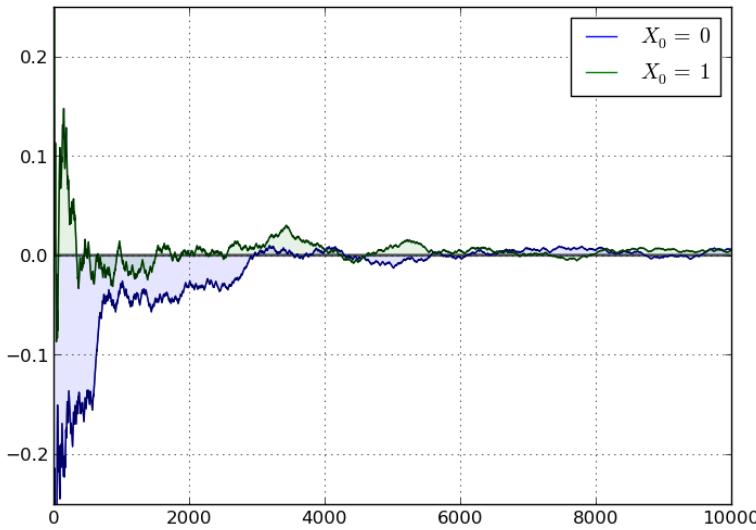
In other words, if  $\{X_t\}$  represents the Markov chain for employment, then  $\bar{X}_n \rightarrow p$  as  $n \rightarrow \infty$ , where

$$\bar{X}_n := \frac{1}{n} \sum_{t=1}^n \mathbf{1}\{X_t = 1\}$$

Your exercise is to illustrate this convergence

First,

- generate one simulated time series  $\{X_t\}$  of length 10,000, starting at  $X_0 = 1$
- plot  $\bar{X}_n - p$  against  $n$ , where  $p$  is as defined above



Second, repeat the first step, but this time taking  $X_0 = 2$

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”

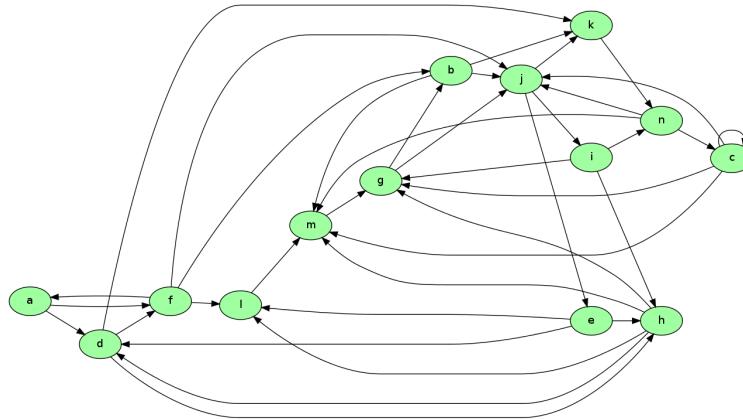
Ranking according to a measure of importance is the problem we now consider

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

```
julia> matchall(r"\w", "x +++ y ***** z")
3-element Array{SubString{UTF8String},1}:
 "x"
 "y"
 "z"

julia> matchall(r"\w", "a ^~ b &&& \$\$ c")
3-element Array{SubString{UTF8String},1}:
 "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

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  $s = (x_{n-1} - x_0)/(n - 1)$
- $P[i, j]$  represents the probability of transitioning from  $x_i$  to  $x_j$

Let  $F$  be the cumulative distribution function of the normal distribution  $N(0, \sigma_u^2)$

The values  $P[i, j]$  are computed to approximate the AR(1) process — omitting the derivation, the rules are as follows:

1. If  $j = 0$ , then set

$$P[i, j] = P[i, 0] = F(x_0 - \rho x_i + s/2)$$

2. If  $j = n - 1$ , then set

$$P[i, j] = P[i, n - 1] = 1 - F(x_{n-1} - \rho x_i - s/2)$$

3. Otherwise, set

$$P[i, j] = F(x_j - \rho x_i + s/2) - F(x_j - \rho x_i - s/2)$$

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

## Solutions

[Solution notebook](#)

## 2.3 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.

For us, the shortest path problem also provides a simple introduction to the logic of dynamic programming, which is one of our key topics

Variations of the methods we discuss are used millions of times every day, in applications such as Google Maps

### 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

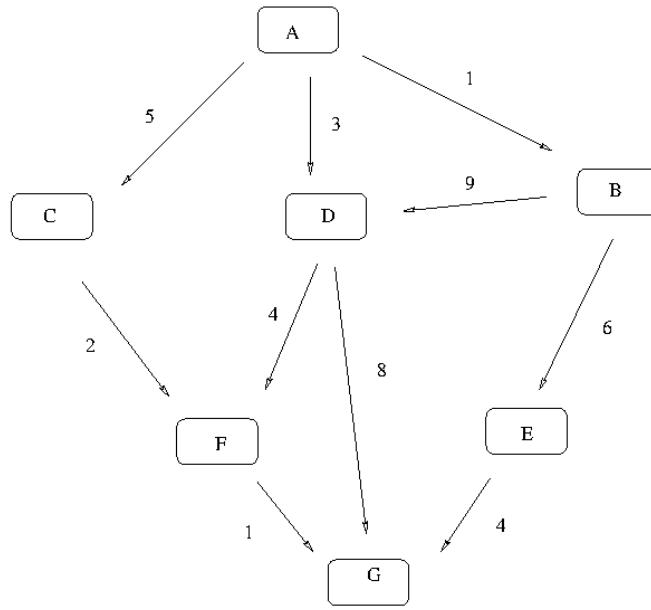
Consider the following graph

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  $J(G) = 0$

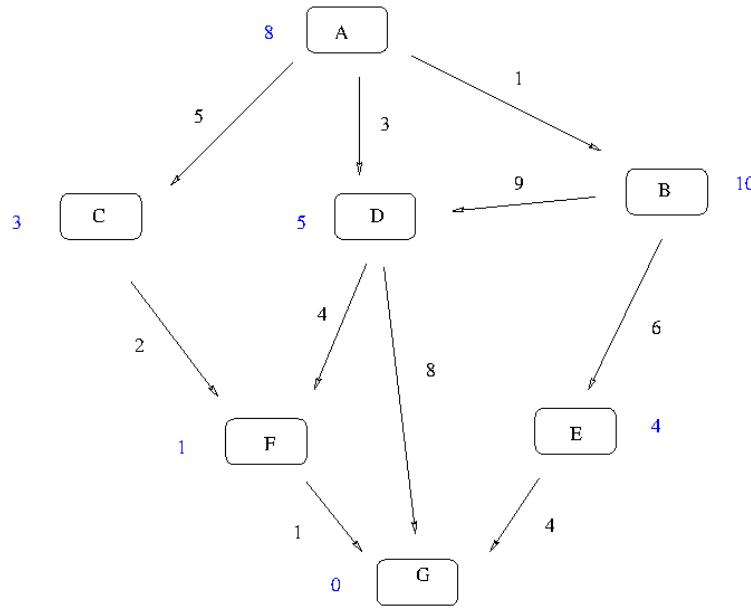
Intuitively, the best path can now be found as follows

- Start at A
- From node  $v$ , move to any node that solves

$$\min_{w \in F_v} \{c(v, w) + J(w)\} \quad (2.14)$$

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

$$J(v) = \min_{w \in F_v} \{c(v, w) + J(w)\} \quad (2.15)$$

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

$$J_0(v) = M \text{ if } v \neq \text{destination, else } J_0(v) = 0 \quad (2.16)$$

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](#)

## 2.4 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

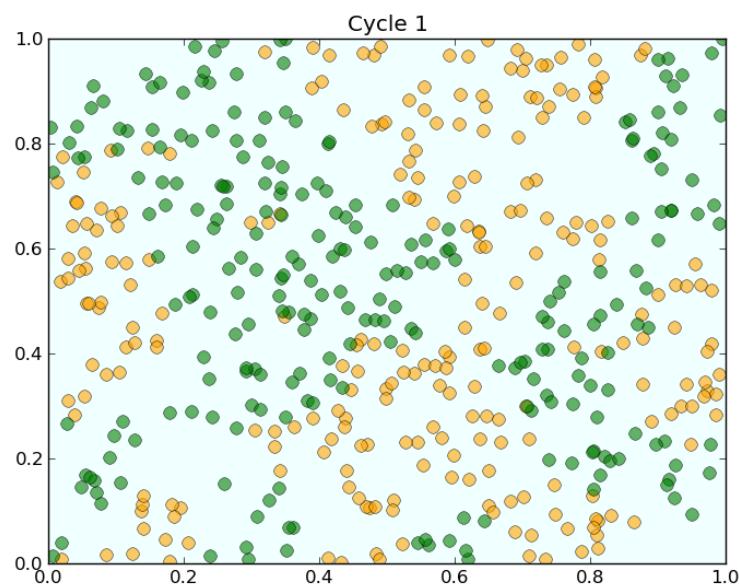
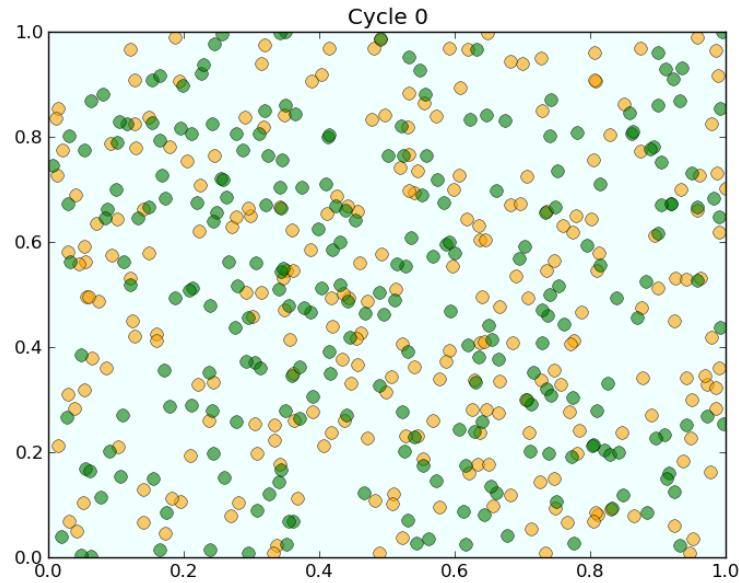
We continue to cycle until no one wishes to move

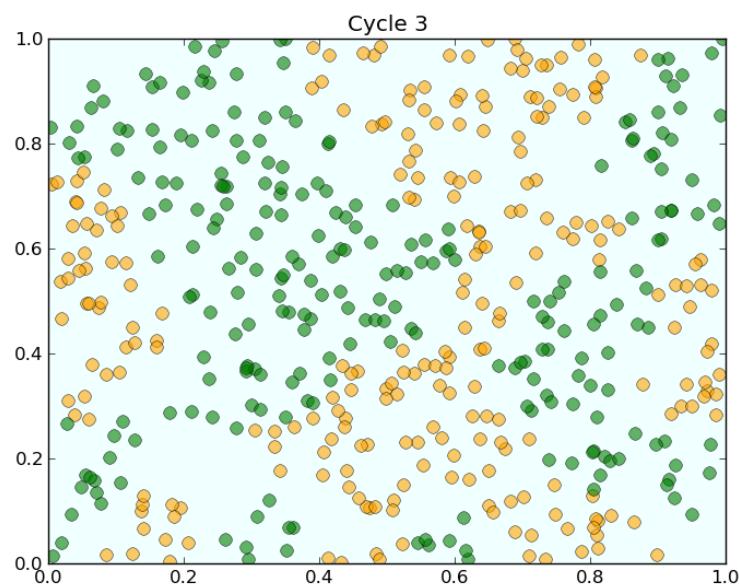
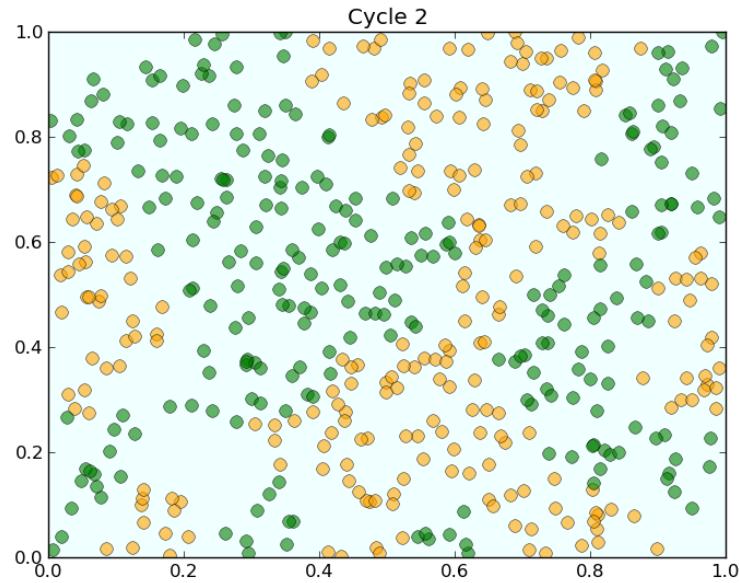
### Results

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
    end
end
```

Use 250 agents of each type

### Solutions

[Solution notebook](#)

## 2.5 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

$$\mathbb{P}\{\bar{X}_n \rightarrow \mu \text{ as } n \rightarrow \infty\} = 1 \quad (2.17)$$

What does this last expression mean?

Let's think about it from a simulation perspective, imagining for a moment that our computer can generate perfect random samples (which of course **it can't**)

Let's also imagine that we can generate infinite sequences, so that the statement  $\bar{X}_n \rightarrow \mu$  can be evaluated

In this setting, (2.17) 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

$$\mathbb{P}\{|\bar{X}_n - \mu| \geq \epsilon\} \rightarrow 0 \quad \text{as } n \rightarrow \infty \quad (2.18)$$

(This version is weaker because we claim only **convergence in probability** rather than **almost sure convergence**, and assume a finite second moment)

To see that this is so, fix  $\epsilon > 0$ , and let  $\sigma^2$  be the variance of each  $X_i$

Recall the **Chebyshev inequality**, which tells us that

$$\mathbb{P}\{|\bar{X}_n - \mu| \geq \epsilon\} \leq \frac{\mathbb{E}[(\bar{X}_n - \mu)^2]}{\epsilon^2} \quad (2.19)$$

Now observe that

$$\begin{aligned}
 \mathbb{E}[(\bar{X}_n - \mu)^2] &= \mathbb{E} \left\{ \left[ \frac{1}{n} \sum_{i=1}^n (X_i - \mu) \right]^2 \right\} \\
 &= \frac{1}{n^2} \sum_{i=1}^n \sum_{j=1}^n \mathbb{E}(X_i - \mu)(X_j - \mu) \\
 &= \frac{1}{n^2} \sum_{i=1}^n \mathbb{E}(X_i - \mu)^2 \\
 &= \frac{\sigma^2}{n}
 \end{aligned}$$

Here the crucial step is at the third equality, which follows from independence

Independence means that if  $i \neq j$ , then the covariance term  $\mathbb{E}(X_i - \mu)(X_j - \mu)$  drops out

As a result,  $n^2 - n$  terms vanish, leading us to a final expression that goes to zero in  $n$

Combining our last result with (2.19), we come to the estimate

$$\mathbb{P}\{| \bar{X}_n - \mu | \geq \epsilon\} \leq \frac{\sigma^2}{n\epsilon^2} \quad (2.20)$$

The claim in (2.18) 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

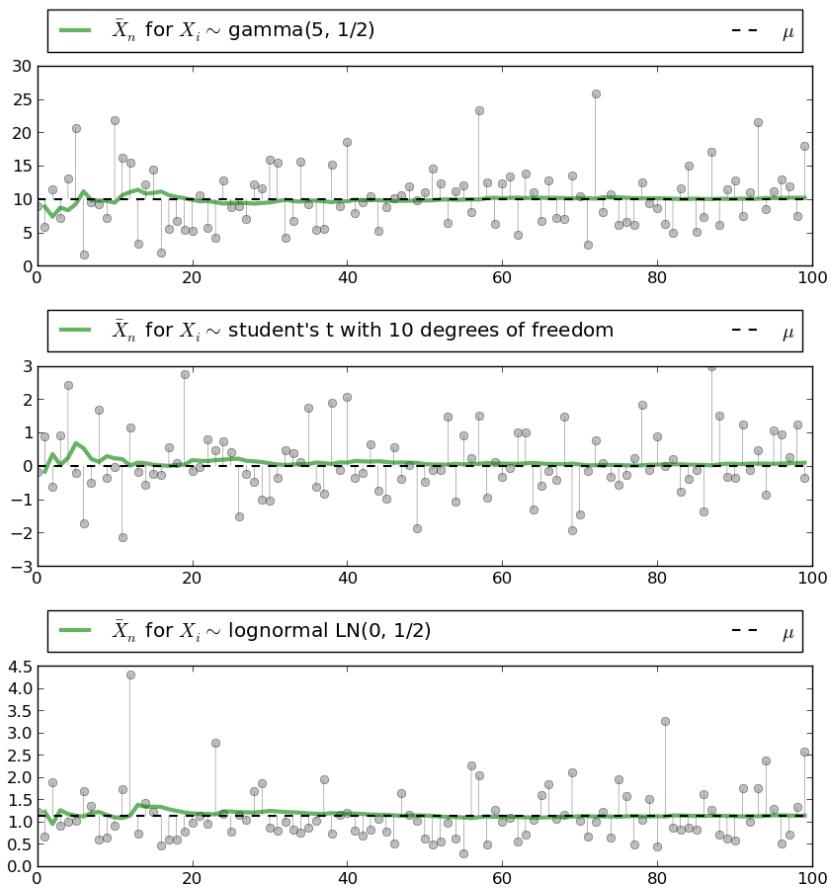
In particular, we aim to generate some sequences of IID random variables and plot the evolution of  $\bar{X}_n$  as  $n$  increases

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 `illuminates_lln.jl`, which is shown below (and can be found in the `examples` directory of the [main repository](#))

The three distributions are chosen at random from a selection stored in the dictionary `distributions`

```
#=
Visual illustration of the law of large numbers.

@author : Spencer Lyon <spencer.lyon@nyu.edu>

References
-----

Based off the original python file illuminates_lln.py
=#
using PyPlot
using Distributions

n = 100
srand(42) # reproducible results

# == Arbitrary collection of distributions == #
distributions = { "student's t with 10 degrees of freedom" => TDist(10),
                  "beta(2, 2)" => Beta(2.0, 2.0),
                  "lognormal LN(0, 1/2)" => LogNormal(0.5),
                  "gamma(5, 1/2)" => Gamma(5.0, 2.0),
                  "poisson(4)" => Poisson(4),
                  "exponential with lambda = 1" => Exponential(1) }

# == Create a figure and some axes == #
num_plots = 3
fig, axes = plt.subplots(num_plots, 1, figsize=(10, 10))

bbox = [0., 1.02, 1., .102]
legend_args = {:ncol => 2,
               :bbox_to_anchor => bbox,
               :loc => 3,
               :mode => "expand"}
subplots_adjust(hspace=0.5)

for ax in axes
    dist_names = collect(keys(distributions))
    # == Choose a randomly selected distribution == #
    name = dist_names[rand(1:length(dist_names))]
    dist = pop!(distributions, name)

    # == Generate n draws from the distribution == #
    data = rand(dist, n)

    # == Compute sample mean at each n == #
    sample_mean = Array(Float64, n)
    for i=1:n
```

```

    sample_mean[i] = mean(data[1:i])
end

# == Plot ==
ax[:plot](1:n, data, "o", color="grey", alpha=0.5)
axlabel = LaTeXString("\$\\bar{X}_n\$ for \$X_i \\sim \$ name")
ax[:plot](1:n, sample_mean, "g-", lw=3, alpha=0.6, label=axlabel)
m = mean(dist)
ax[:plot](1:n, ones(n)*m, "k--", lw=1.5, label=L"\$\\mu\$")
ax[:vlines](1:n, m, data, lw=0.2)
ax[:legend](;legend_args...)
end

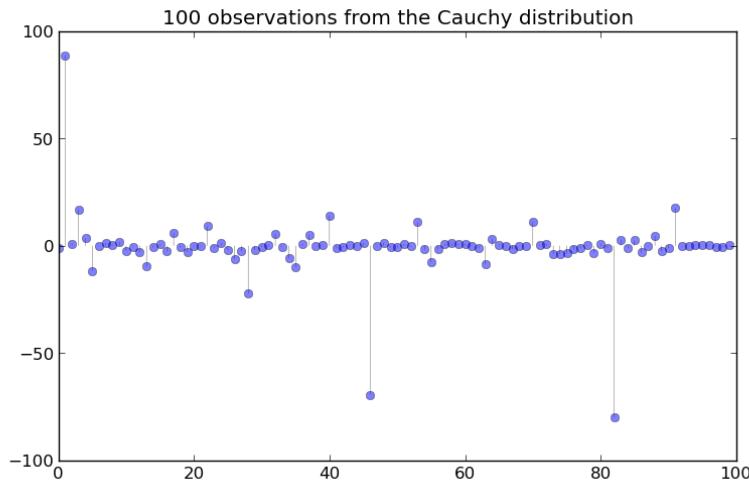
```

**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



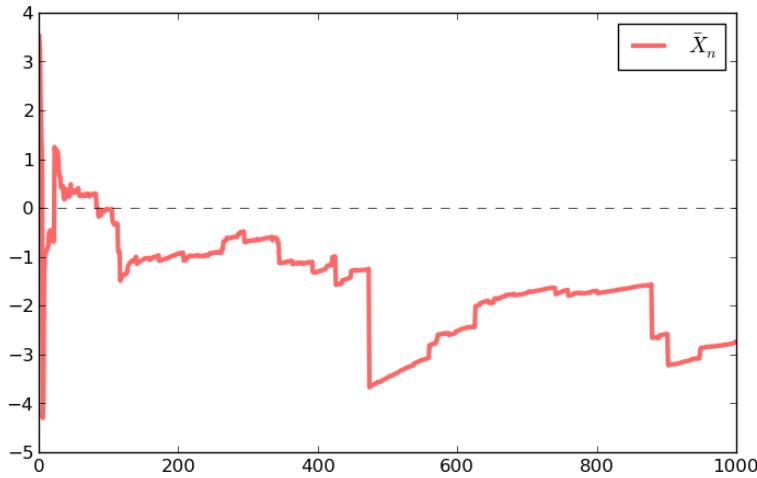
Notice how extreme observations are far more prevalent here than the previous figure

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

$$\phi(t) = \mathbb{E}e^{itX} = \int e^{itx}f(x)dx = e^{-|t|} \quad (2.21)$$

Using independence, the characteristic function of the sample mean becomes

$$\begin{aligned} \mathbb{E}e^{it\bar{X}_n} &= \mathbb{E}\exp\left\{i\frac{t}{n}\sum_{j=1}^n X_j\right\} \\ &= \mathbb{E}\prod_{j=1}^n \exp\left\{i\frac{t}{n}X_j\right\} \\ &= \prod_{j=1}^n \mathbb{E}\exp\left\{i\frac{t}{n}X_j\right\} = [\phi(t/n)]^n \end{aligned}$$

In view of (2.21), this is just  $e^{-|t|}$

Thus, in the case of the Cauchy distribution, the sample mean itself has the very same Cauchy distribution, regardless of  $n$

In particular, the sequence  $\bar{X}_n$  does not converge to a point

## CLT

Next we turn to the central limit theorem, which tells us about the distribution of the deviation between sample averages and population means

**Statement of the Theorem** The central limit theorem is one of the most remarkable results in all of mathematics

In the classical IID setting, it tells us the following: If the sequence  $X_1, \dots, X_n$  is IID, with common mean  $\mu$  and common variance  $\sigma^2 \in (0, \infty)$ , then

$$\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} N(0, \sigma^2) \quad \text{as } n \rightarrow \infty \quad (2.22)$$

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

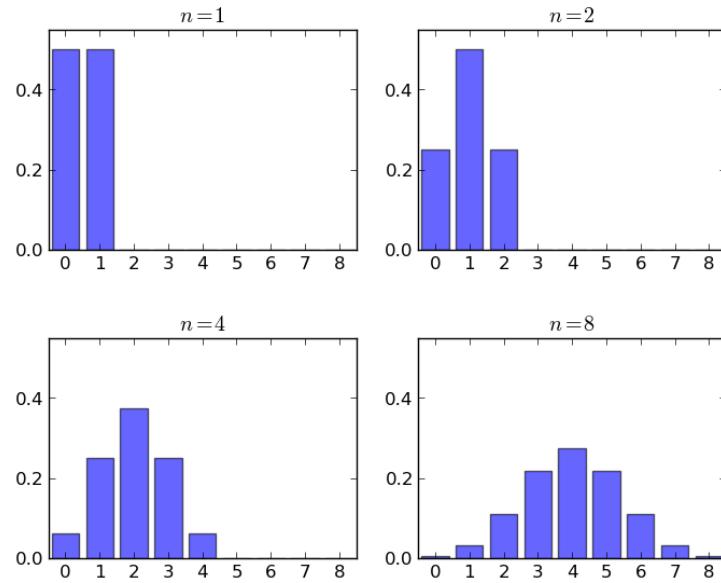
Why does adding independent copies produce a bell-shaped distribution?

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)

```
#=
Visual illustration of the central limit theorem

@author : Spencer Lyon <spencer.lyon@nyu.edu>

References
-----

Based off the original python file illustrates_clt.py
=#
using PyPlot
using Distributions

# == Set parameters == #
```

```

strand(42) # reproducible results
n = 250    # Choice of n
k = 100000  # Number of draws of  $Y_n$ 
dist = Exponential(1./2.) # Exponential distribution, lambda = 1/2
mu, s = mean(dist), std(dist)

# == Draw underlying RVs. Each row contains a draw of  $X_1, \dots, X_n$  ==
data = rand(dist, (k, n))

# == Compute mean of each row, producing k draws of  $\bar{X}_n$  ==
sample_means = mean(data, 2)

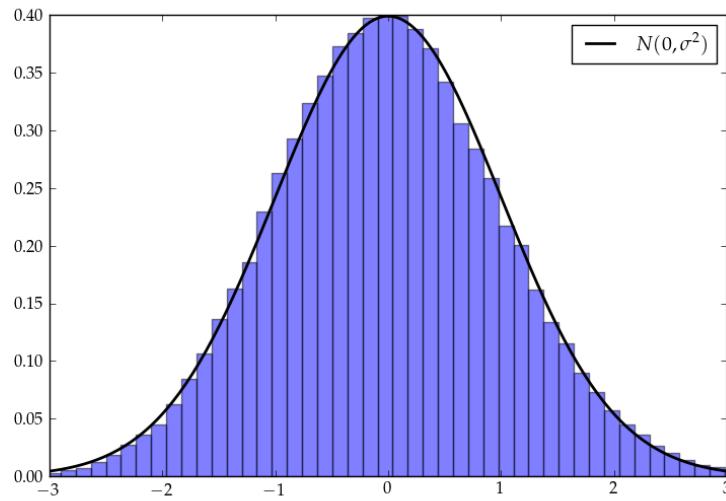
# == Generate observations of  $Y_n$  ==
Y = sqrt(n) * (sample_means .- mu)

# == Plot ==
fig, ax = subplots()
xmin, xmax = -3 * s, 3 * s
ax[:set_xlim](xmin, xmax)
ax[:hist](Y, bins=60, alpha=0.5, normed=true)
xgrid = linspace(xmin, xmax, 200)
ax[:plot](xgrid, pdf(Normal(0.0, s), xgrid), "k-", lw=2,
           label=LaTeXString("\$N(0, \sigma^2)\$"))
ax[:legend]()

```

The file is `illustiates_clt.jl`, from the [main repository](#)

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`

**Simulation 2** Our next simulation is somewhat like the first, except that we aim to track the distribution of  $Y_n := \sqrt{n}(X_n - \mu)$  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

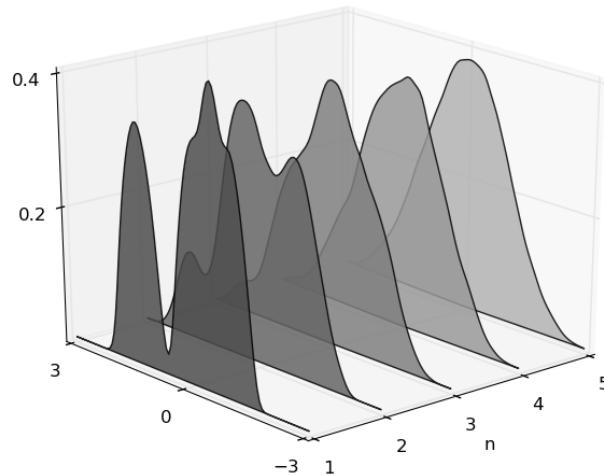
For  $n = 2$ , the distribution of  $Y_2$  is that of  $(X_1 + X_2)/\sqrt{2}$ , and so on

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 `examples/clt3d.jl`, which is available from the [main repository](#)

We leave you to investigate its contents if you wish to know more

If you run the file from the ordinary Julia or IJulia 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

$$\text{Var}[\mathbf{X}] = \begin{pmatrix} \mathbb{E}[(X_1 - \mu_1)(X_1 - \mu_1)] & \dots & \mathbb{E}[(X_1 - \mu_1)(X_k - \mu_k)] \\ \mathbb{E}[(X_2 - \mu_2)(X_1 - \mu_1)] & \dots & \mathbb{E}[(X_2 - \mu_2)(X_k - \mu_k)] \\ \vdots & \vdots & \vdots \\ \mathbb{E}[(X_k - \mu_k)(X_1 - \mu_1)] & \dots & \mathbb{E}[(X_k - \mu_k)(X_k - \mu_k)] \end{pmatrix}$$

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

$$\mathbb{P}\{\bar{\mathbf{X}}_n \rightarrow \boldsymbol{\mu} \text{ as } n \rightarrow \infty\} = 1 \quad (2.23)$$

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,

$$\sqrt{n}(\bar{\mathbf{X}}_n - \boldsymbol{\mu}) \xrightarrow{d} N(\mathbf{0}, \Sigma) \quad \text{as } n \rightarrow \infty \quad (2.24)$$

### Exercises

**Exercise 1** One very useful consequence of the central limit theorem is as follows

Assume the conditions of the CLT as *stated above*

If  $g: \mathbb{R} \rightarrow \mathbb{R}$  is differentiable at  $\mu$  and  $g'(\mu) \neq 0$ , then

$$\sqrt{n}\{g(\bar{X}_n) - g(\mu)\} \xrightarrow{d} N(0, g'(\mu)^2\sigma^2) \quad \text{as } n \rightarrow \infty \quad (2.25)$$

This theorem is used frequently in statistics to obtain the asymptotic distribution of estimators — many of which can be expressed as functions of sample means

(These kinds of results are often said to use the “delta method”)

The proof is based on a Taylor expansion of  $g$  around the point  $\mu$

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.jl` 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

$$\sqrt{n}(\bar{\mathbf{X}}_n - \boldsymbol{\mu}) \xrightarrow{d} N(\mathbf{0}, \Sigma) \quad (2.26)$$

is valid

In a statistical setting, one often wants the right hand side to be **standard** normal, so that confidence intervals are easily computed

This normalization can be achieved on the basis of three observations

First, if  $\mathbf{X}$  is a random vector in  $\mathbb{R}^k$  and  $\mathbf{A}$  is constant and  $k \times k$ , then

$$\text{Var}[\mathbf{AX}] = \mathbf{A} \text{Var}[\mathbf{X}] \mathbf{A}'$$

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

$$\mathbf{Q}\mathbf{S}\mathbf{Q}' = \mathbf{I}$$

Here  $\mathbf{I}$  is the  $k \times k$  identity matrix

Putting these things together, your first exercise is to show that if  $\mathbf{Q}$  is the inverse square root of  $\Sigma$ , 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

$$n\|\mathbf{Q}(\bar{\mathbf{X}}_n - \boldsymbol{\mu})\|^2 \xrightarrow{d} \chi^2(k) \quad (2.27)$$

where  $\chi^2(k)$  is the chi-squared distribution with  $k$  degrees of freedom

(Recall that  $k$  is the dimension of  $\mathbf{X}_i$ , the underlying random vectors)

Your second exercise is to illustrate the convergence in (2.27) with a simulation

In doing so, let

$$\mathbf{x}_i := \begin{pmatrix} W_i \\ U_i + W_i \end{pmatrix}$$

where

- each  $W_i$  is an IID draw from the uniform distribution on  $[-1, 1]$
- each  $U_i$  is an IID draw from the uniform distribution on  $[-2, 2]$
- $U_i$  and  $W_i$  are independent of each other

Hints:

1. `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](#)

## 2.6 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

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t \\ x_0 &\sim N(\mu_0, \Sigma_0) \end{aligned} \tag{2.28}$$

**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 (2.28) pins down the values of the sequences  $\{x_t\}$  and  $\{y_t\}$

Even without these draws, the primitives 1–3 pin down the *probability distributions* of  $\{x_t\}$  and  $\{y_t\}$

Later we'll see how to compute these distributions and their moments

**Martingale difference shocks** We've made the common assumption that the shocks are independent standardized normal vectors

But some of what we say will 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

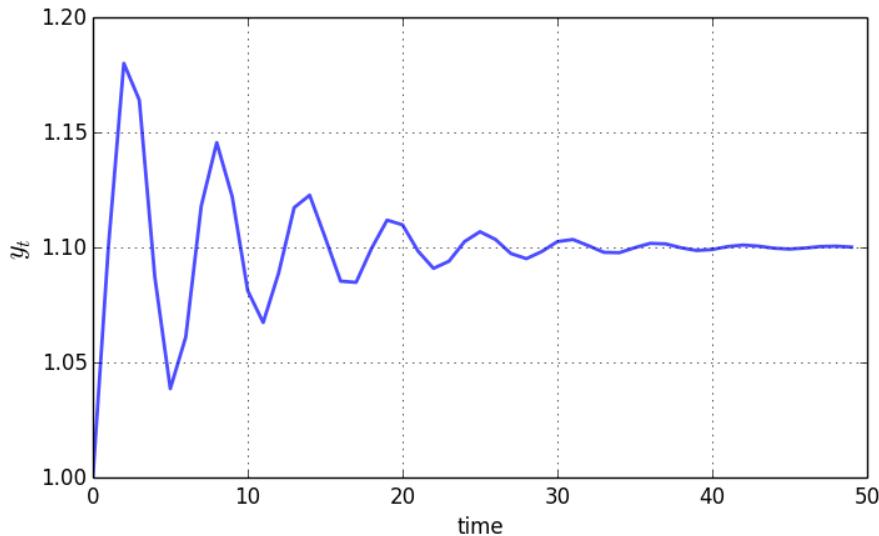
$$y_{t+1} = \phi_0 + \phi_1 y_t + \phi_2 y_{t-1} \quad \text{s.t. } y_0, y_{-1} \text{ given} \quad (2.29)$$

To map (2.29) into our state space system (2.28), 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, (2.28) and (2.29) 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 (2.28) to represent the model

$$y_{t+1} = \phi_1 y_t + \phi_2 y_{t-1} + \phi_3 y_{t-2} + \phi_4 y_{t-3} + \sigma w_{t+1} \quad (2.30)$$

where  $\{w_t\}$  is iid and standard normal

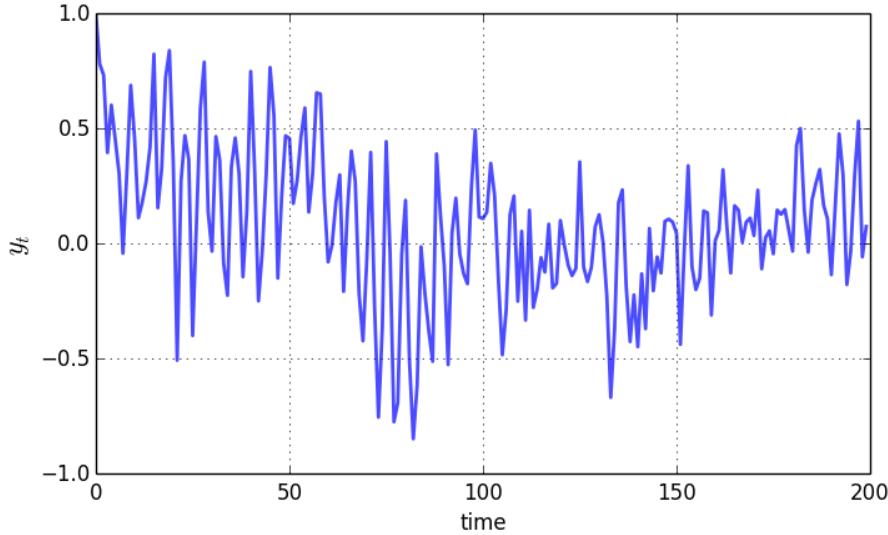
To put this in the linear state space format we take  $x_t = [y_t \ y_{t-1} \ y_{t-2} \ y_{t-3}]'$  and

$$A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad G = [1 \ 0 \ 0 \ 0]$$

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 (2.30) is termed a *vector autoregression*

To map this into (2.28), we set

$$x_t = \begin{bmatrix} y_t \\ y_{t-1} \\ y_{t-2} \\ y_{t-3} \end{bmatrix} \quad A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \end{bmatrix} \quad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad G = [I \ 0 \ 0 \ 0]$$

where  $I$  is the  $k \times k$  identity matrix and  $\sigma$  is a  $k \times k$  matrix

**Seasonals** We can use (2.28) to represent

1. the *deterministic seasonal*  $y_t = y_{t-4}$
2. the *indeterministic seasonal*  $y_t = \phi_4 y_{t-4} + w_t$

In fact both are special cases of (2.30)

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>3</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

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad G = [a \ b] \quad (2.31)$$

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  $y_t = t + b + \sum_{j=0}^t w_j$  with  $w_0 = 0$

To modify (2.31) accordingly, we set

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad G = [1 \ b] \quad (2.32)$$

For reasons explained below,  $y_t$  is called a *martingale with drift*

---

<sup>3</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 (2.28) repeatedly to obtain

$$\begin{aligned} x_t &= Ax_{t-1} + Cw_t \\ &= A^2x_{t-2} + ACw_{t-1} + Cw_t \\ &\quad \vdots \\ &= \sum_{j=0}^{t-1} A^j C w_{t-j} + A^t x_0 \end{aligned} \tag{2.33}$$

Representation (2.33) 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, recall the model (2.32)

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 (2.33), 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 (2.28), it's easy to obtain expressions for the (unconditional) mean of  $x_t$  and  $y_t$

We'll explain what *unconditional* and *conditional* mean soon

Letting  $\mu_t := \mathbb{E}[x_t]$  and using linearity of expectations, we find that

$$\mu_{t+1} = A\mu_t \quad \text{with} \quad \mu_0 \text{ given} \tag{2.34}$$

Here  $\mu_0$  is a primitive given in (2.28)

The variance-covariance matrix of  $x_t$  is  $\Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)']$

Using  $x_{t+1} - \mu_{t+1} = A(x_t - \mu_t) + Cw_{t+1}$ , we can determine this matrix recursively via

$$\Sigma_{t+1} = A\Sigma_t A' + CC' \quad \text{with } \Sigma_0 \text{ given} \quad (2.35)$$

As with  $\mu_0$ , the matrix  $\Sigma_0$  is a primitive given in (2.28)

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

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t] = G\mu_t \quad (2.36)$$

The variance-covariance matrix of  $y_t$  is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t] = G\Sigma_t G' \quad (2.37)$$

**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

$$u \sim N(\bar{u}, S) \quad \text{and} \quad v = a + Bu \implies v \sim N(a + B\bar{u}, BSB') \quad (2.38)$$

In particular, given our Gaussian assumptions on the primitives and the linearity of (2.28) we can see immediately that both  $x_t$  and  $y_t$  are Gaussian for all  $t \geq 0$ <sup>4</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 (2.34) and (2.35)

---

<sup>4</sup> The correct way to argue this is by induction. Suppose that  $x_t$  is Gaussian. Then (2.28) and (2.38) 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

$$x_t \sim N(\mu_t, \Sigma_t) \quad (2.39)$$

By similar reasoning combined with (2.36) and (2.37),

$$y_t \sim N(G\mu_t, G\Sigma_t G') \quad (2.40)$$

**Ensemble Interpretations** How should we interpret the distributions defined by (2.39)–(2.40)?

Intuitively, the probabilities in a distribution correspond to relative frequencies in a large population drawn from that distribution

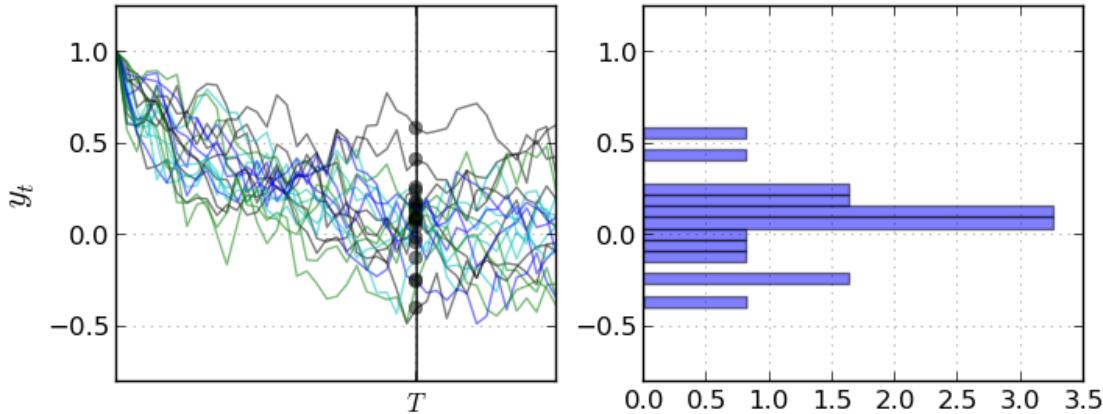
Let's apply this idea to our setting, focusing on the distribution of  $y_T$  for fixed  $T$

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 (2.30)

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 `examples/paths_and_hist.jl` from the [main repository](#))

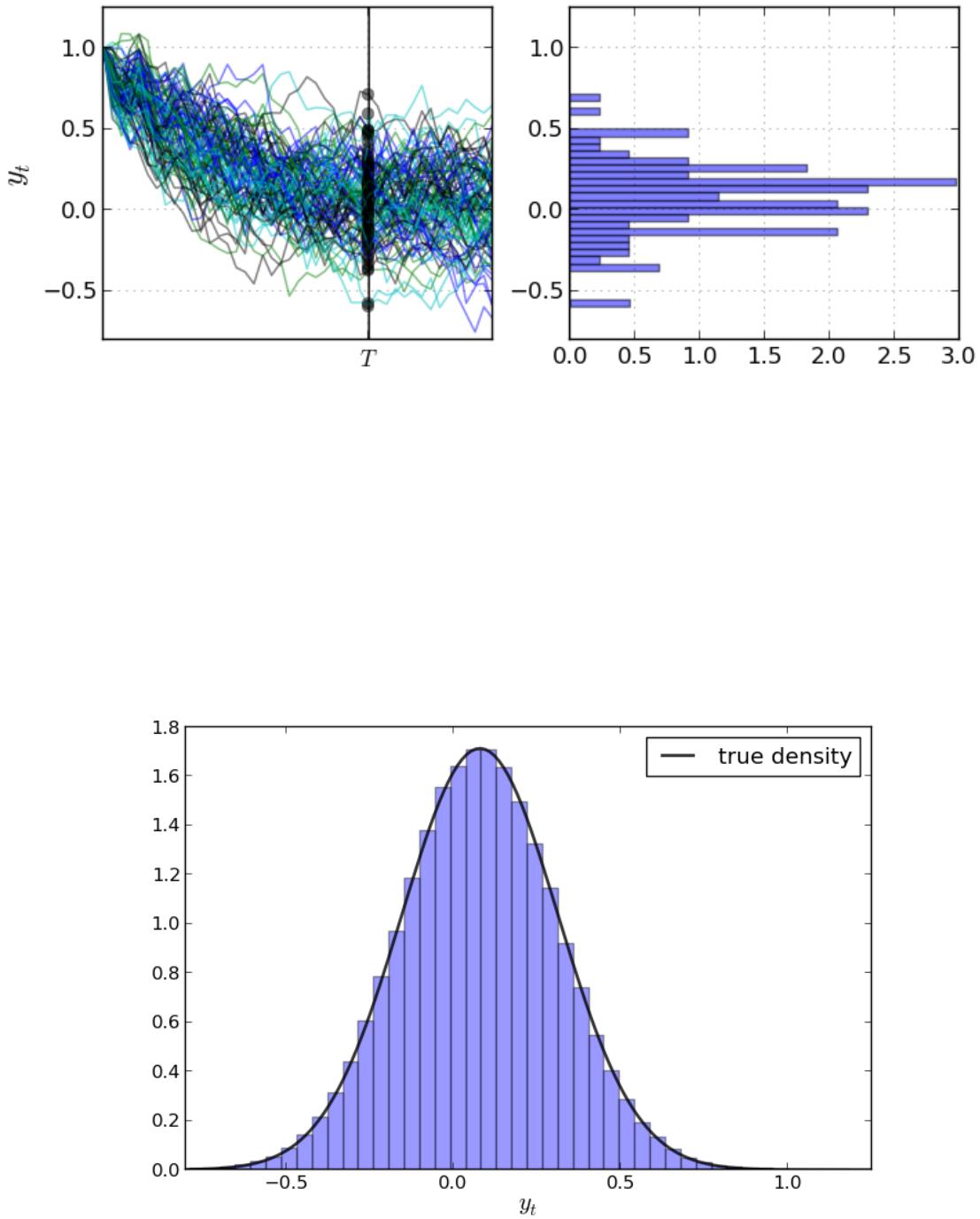
Here is another figure, this time with 100 observations

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 (2.40)

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

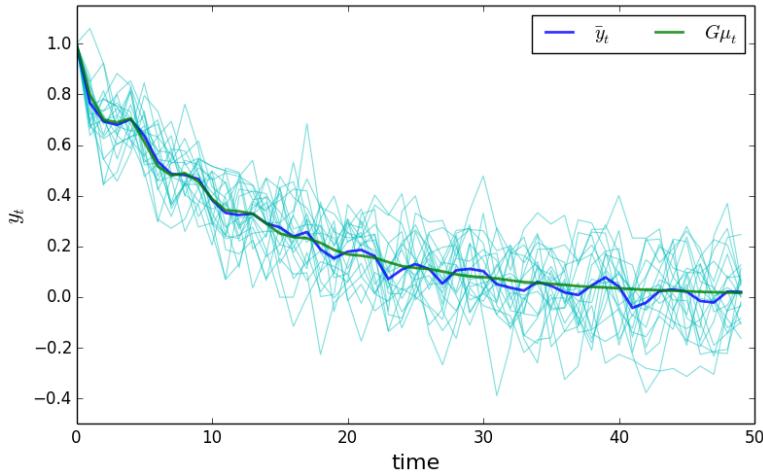
Just as the histogram corresponds to the distribution, the *ensemble* or *cross-sectional average*

$$\bar{y}_T := \frac{1}{I} \sum_{i=1}^I y_T^i$$

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 right-hand side  $\mu_T$  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 in general 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 (2.28), the conditional densities are

$$p(x_{t+1} | x_t) = N(Ax_t, CC')$$

**Autocovariance functions** An important object related to the joint distribution is the *autocovariance function*

$$\Sigma_{t+j,t} := \mathbb{E} [(x_{t+j} - \mu_{t+j})(x_t - \mu_t)'] \quad (2.41)$$

Elementary calculations show that

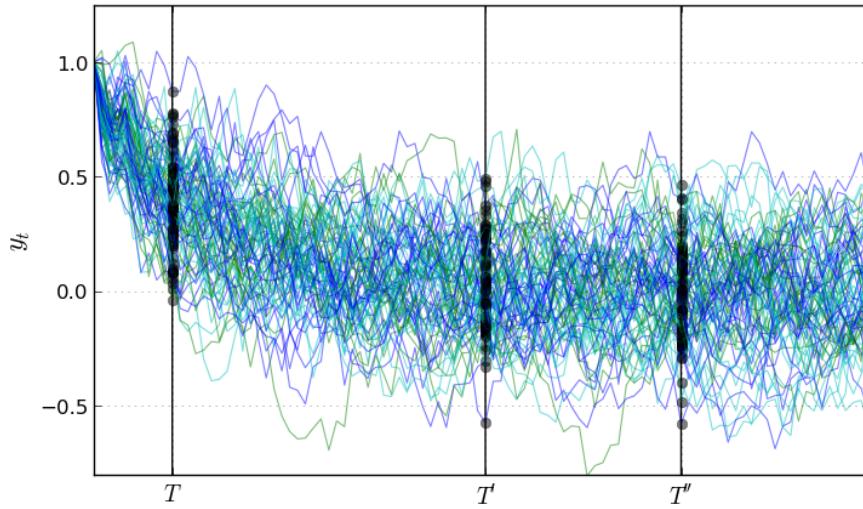
$$\Sigma_{t+j,t} = A^j \Sigma_t \quad (2.42)$$

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

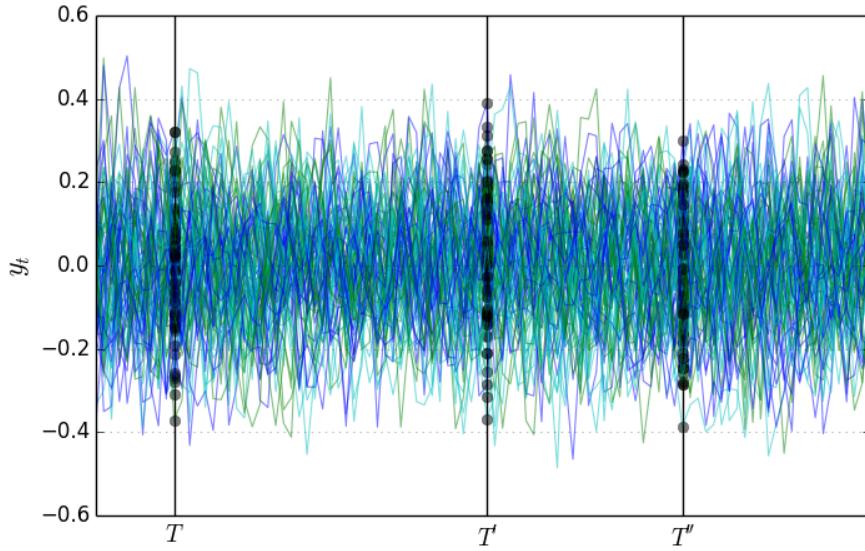
$$\psi_\infty = N(\mu_\infty, \Sigma_\infty)$$

where  $\mu_\infty$  and  $\Sigma_\infty$  are fixed points of (2.34) and (2.35) 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 (2.34) and (2.35) 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 (2.42), 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 `all(abs(eigvals(A)) < 1) == true`

The difference equation (2.35) 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

$$\mu_{1,t+1} = A_1 \mu_{1,t} + a \tag{2.43}$$

Assume now that the moduli of the eigenvalues of  $A_1$  are all strictly less than one

Then (2.43) 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

$$\begin{aligned} \Sigma_\infty &= A \Sigma_\infty A' + C C' \\ \Sigma_{t+j,t} &= A^j \Sigma_\infty \end{aligned} \tag{2.44}$$

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 (2.35) converge to the fixed point of the *discrete Lyapunov equation* in the first line of (2.44)

---

**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,

- $\frac{1}{T} \sum_{t=0}^T x_t \rightarrow \mu_\infty$
- $\frac{1}{T} \sum_{t=0}^T (x_t - \bar{x}_T)(x_t - \bar{x}_T)' \rightarrow \Sigma_\infty$
- $\frac{1}{T} \sum_{t=0}^T (x_{t+j} - \bar{x}_T)(x_t - \bar{x}_T)' \rightarrow A^j \Sigma_\infty$

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

$$\begin{aligned}x_{t+1} &= Ax_t + Cw_{t+1} \\y_t &= Gx_t + Hv_t \\x_0 &\sim N(\mu_0, \Sigma_0)\end{aligned}\tag{2.45}$$

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 (2.36) and (2.37) now become

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t + Hv_t] = G\mu_t\tag{2.46}$$

The variance-covariance matrix of  $y_t$  is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t + Hv_t] = G\Sigma_t G' + HH'\tag{2.47}$$

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

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

$$x_{t+j} - \mathbb{E}_t[x_{t+j}] = \sum_{s=0}^{j-1} A^s C w_{t-s+j} \quad (2.48)$$

Evidently,

$$V_j := \mathbb{E}_t[(x_{t+j} - \mathbb{E}_t[x_{t+j}]) (x_{t+j} - \mathbb{E}_t[x_{t+j}])'] = \sum_{k=0}^{j-1} A^k C C' A^k' \quad (2.49)$$

$V_j$  defined in (2.49) can be calculated recursively via  $V_1 = CC'$  and

$$V_j = CC' + AV_{j-1}A', \quad j \geq 2 \quad (2.50)$$

$V_j$  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

$$V_\infty = CC' + AV_\infty A' \quad (2.51)$$

Equation (2.51) 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 (2.28)

We want the following objects

- Forecast of a geometric sum of future  $x$ 's, or  $\mathbb{E}_t \left[ \sum_{j=0}^{\infty} \beta^j x_{t+j} \right]$
- Forecast of a geometric sum of future  $y$ 's, or  $\mathbb{E}_t \left[ \sum_{j=0}^{\infty} \beta^j y_{t+j} \right]$

These objects are important components of some famous and interesting dynamic models

For example,

- if  $\{y_t\}$  is a stream of dividends, then  $\mathbb{E} \left[ \sum_{j=0}^{\infty} \beta^j y_{t+j} | x_t \right]$  is a model of a stock price
- if  $\{y_t\}$  is the money supply, then  $\mathbb{E} \left[ \sum_{j=0}^{\infty} \beta^j y_{t+j} | x_t \right]$  is a model of the price level

**Formulas** Fortunately, it is easy to use a little matrix algebra to compute these objects

Suppose that every eigenvalue of  $A$  has modulus strictly less than  $\frac{1}{\beta}$

It *then follows* that  $I + \beta A + \beta^2 A^2 + \dots = [I - \beta A]^{-1}$

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.jl` from the `QuantEcon` package

The code implements a type for handling linear state space models (simulations, calculating moments, etc.)

We repeat it here for convenience

```
#=
Computes quantities related to the Gaussian linear state space model

x_{t+1} = A x_t + C w_{t+1}

y_t = G x_t

The shocks {w_t} are iid and N(0, I)

@author : Spencer Lyon <spencer.lyon@nyu.edu>
@date : 2014-07-28

References
-----
```

```

TODO: Come back and update to match `LinearStateSpace` type from py side
TODO: Add docstrings

http://quant-econ.net/jl/linear\_models.html

=#
import Distributions: MultivariateNormal, rand

#=
numpy allows its multivariate_normal function to have a matrix of
zeros for the covariance matrix; Stats.jl doesn't. This type just
gives a `rand` method when we pass in a matrix of zeros for Sigma_0
so the rest of the api can work, unaffected

The behavior of `rand` is to just pass back the mean vector when
the covariance matrix is zero.
=#
type FakeMVTNorm{T <: Real}
    mu_0::Array{T}
    Sigma_0::Array{T}
end

Base.rand{T}(d::FakeMVTNorm{T}) = copy(d.mu_0)

type LSS
    A::Matrix
    C::Matrix
    G::Matrix
    k::Int
    n::Int
    m::Int
    mu_0::Vector
    Sigma_0::Matrix
    dist::Union(MultivariateNormal, FakeMVTNorm)
end

function LSS(A::ScalarOrArray, C::ScalarOrArray, G::ScalarOrArray,
            mu_0::ScalarOrArray=zeros(size(G, 2)),
            Sigma_0::Matrix=zeros(size(G, 2), size(G, 2)))
    k = size(G, 1)
    n = size(G, 2)
    m = size(C, 2)

    # coerce shapes
    A = reshape([A], n, n)
    C = reshape([C], n, m)
    G = reshape([G], k, n)

    mu_0 = reshape([mu_0], n)

    # define distribution
    if all(Sigma_0 .== 0.0)      # no variance -- no distribution

```

```

        dist = FakeMVTNorm(mu_0, Sigma_0)
    else
        dist = MultivariateNormal(mu_0, Sigma_0)
    end
    LSS(A, C, G, k, n, m, mu_0, Sigma_0, dist)
end

# make kwarg version
function LSS(A::Matrix, C::Matrix, G::Matrix;
            mu_0::Vector=zeros(size(G, 2)),
            Sigma_0::Matrix=zeros(size(G, 2), size(G, 2)))
    return LSS(A, C, G, mu_0, Sigma_0)
end

function simulate(lss::LSS, ts_length=100)
    x = Array(Float64, lss.n, ts_length)
    x[:, 1] = rand(lss.dist)
    w = randn(lss.m, ts_length - 1)
    for t=1:ts_length-1
        x[:, t+1] = lss.A * x[:, t] .+ lss.C * w[:, t]
    end
    y = lss.G * x

    return x, y
end

function replicate(lss::LSS, t=10, num_reps=100)
    x = Array(Float64, lss.n, num_reps)
    for j=1:num_reps
        x_t, _ = simulate(lss, t+1)
        x[:, j] = x_t[:, end]
    end

    y = lss.G * x
    return x, y
end

replicate(lss::LSS; t=10, num_reps=100) = replicate(lss, t, num_reps)

function moment_sequence(lss::LSS)
    A, C, G = lss.A, lss.C, lss.G
    mu_x, Sigma_x = copy(lss.mu_0), copy(lss.Sigma_0)
    while true
        mu_y, Sigma_y = G * mu_x, G * Sigma_x * G'
        produce((mu_x, mu_y, Sigma_x, Sigma_y))

        # Update moments of x
        mu_x = A * mu_x
        Sigma_x = A * Sigma_x * A' + C * C'
    end
end

```

```

    nothing
end

function stationary_distributions(lss::LSS; max_iter=200, tol=1e-5)
    # Initialize iteration
    m = @task moment_sequence(lss)
    mu_x, mu_y, Sigma_x, Sigma_y = consume(m)

    i = 0
    err = tol + 1.

    while err > tol
        if i > max_iter
            println("Convergence failed after $i iterations")
            break
        else
            i += 1
            mu_x1, mu_y, Sigma_x1, Sigma_y = consume(m)
            err_mu = Base.maxabs(mu_x1 - mu_x)
            err_Sigma = Base.maxabs(Sigma_x1 - Sigma_x)
            err = max(err_Sigma, err_mu)
            mu_x, Sigma_x = mu_x1, Sigma_x1
        end
    end

    return mu_x, mu_y, Sigma_x, Sigma_y
end

function geometric_sums(lss::LSS, bet, x_t)
    I = eye(lss.n)
    S_x = (I - bet .* A) \ x_t
    S_y = lss.G * S_x
    return S_x, S_y
end

```

Hopefully the code is relatively self explanatory and adequately documented

Examples of usage are given in the solutions to the exercises

### Exercises

**Exercise 1** Replicate [this figure](#) using the `LinearStateSpace` type from `lss.jl`

**Exercise 2** Replicate [this figure](#) modulo randomness using the same type

**Exercise 3** Replicate [this figure](#) modulo randomness using the same type

The state space model and parameters are the same as for the preceding exercise

**Exercise 4** Replicate *this figure* modulo randomness using the same type

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](#)

## 2.7 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

$$p = N(\hat{x}, \Sigma) \quad (2.52)$$

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

$$\hat{x} = \begin{pmatrix} 0.2 \\ -0.2 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} 0.4 & 0.3 \\ 0.3 & 0.45 \end{pmatrix} \quad (2.53)$$

This density  $p(x)$  is shown below as a contour map, with the center of the red ellipse being equal to  $\hat{x}$

**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

$$y = Gx + v, \quad \text{where } v \sim N(0, R) \quad (2.54)$$

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?

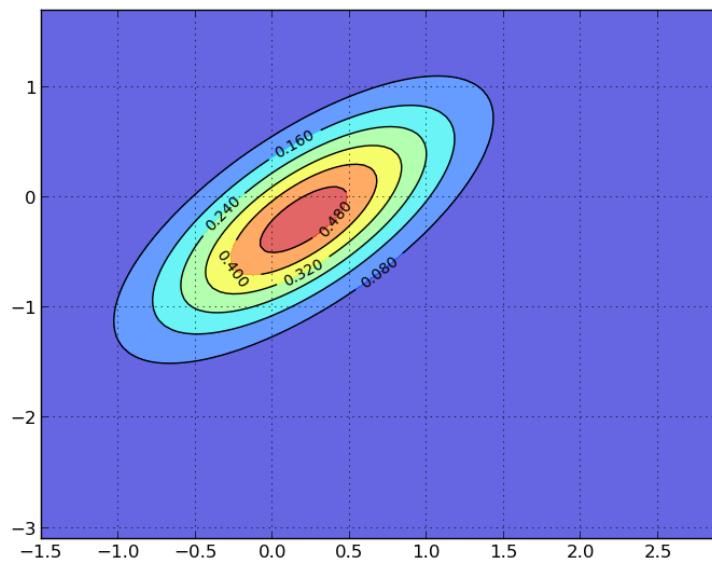
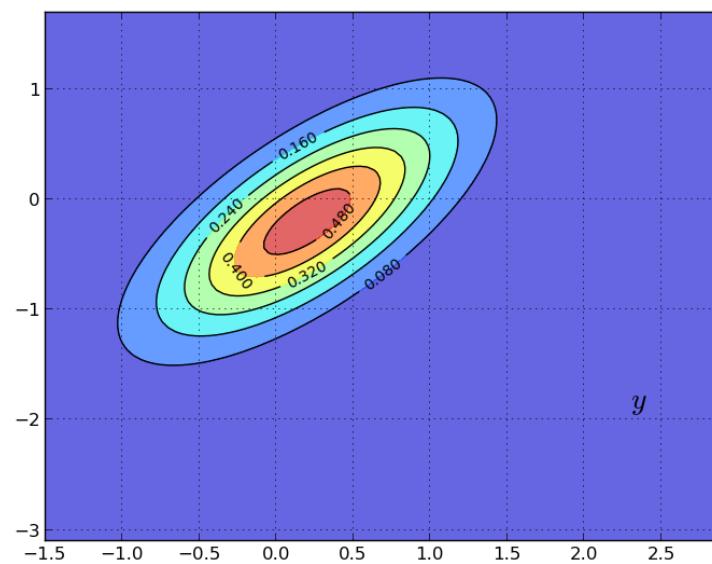


Fig. 2.1: Prior density (Click this or any other figure to enlarge.)



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 (2.54), 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>5</sup> to be

$$p(x | y) = N(\hat{x}^F, \Sigma^F)$$

where

$$\hat{x}^F := \hat{x} + \Sigma G' (G \Sigma G' + R)^{-1} (y - G\hat{x}) \quad \text{and} \quad \Sigma^F := \Sigma - \Sigma G' (G \Sigma G' + R)^{-1} G \Sigma \quad (2.55)$$

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 (2.53)

(The code for generating this and the proceeding figures can be found in the file `gaussian_contours.jl` from the `QuantEcon` package

### The Forecast Step

What have we achieved so far?

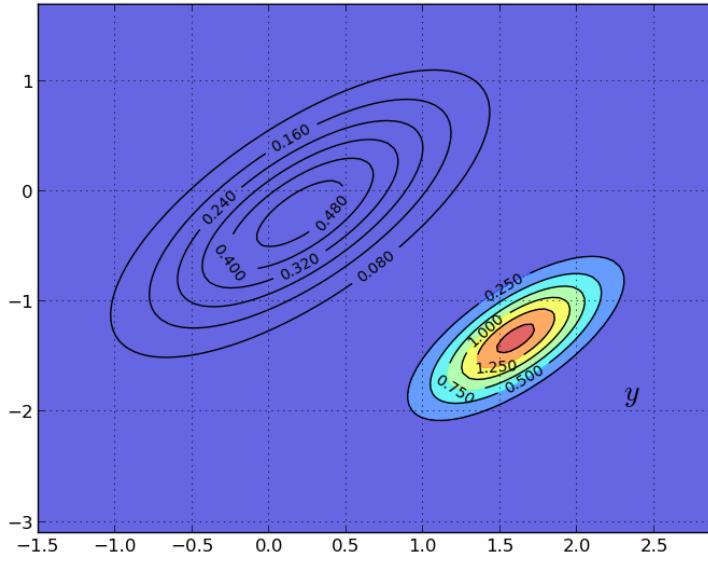
We have obtained probabilities for the current location of the state (missile) given prior and current information

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*

---

<sup>5</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](#).



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,

$$x_{t+1} = Ax_t + w_{t+1}, \quad \text{where } w_t \sim N(0, Q) \quad (2.56)$$

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 (2.56), 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 (2.55) 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  $\hat{x}$

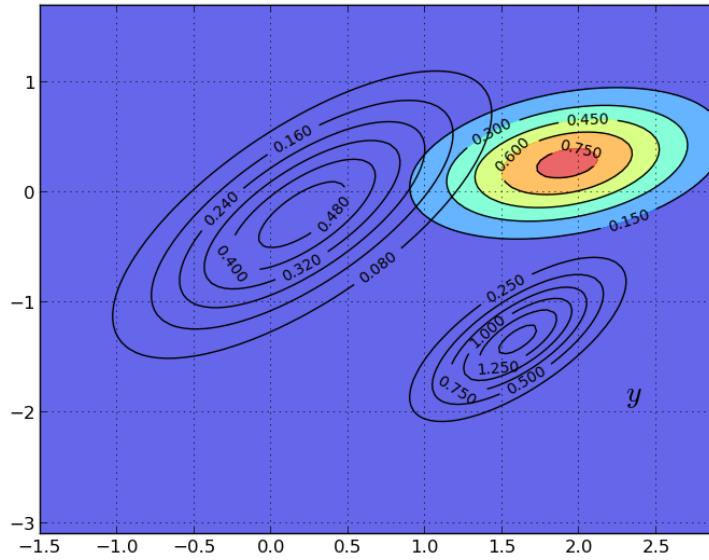
Using this notation, we can summarize our results as follows: Our updated prediction is the density  $N(\hat{x}_{new}, \Sigma_{new})$  where

$$\begin{aligned} \hat{x}_{new} &:= A\hat{x} + K_\Sigma(y - G\hat{x}) \\ \Sigma_{new} &:= A\Sigma A' - K_\Sigma G\Sigma A' + Q \end{aligned} \quad (2.57)$$

- 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 (2.56) 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 (2.54)
4. Compute the predictive distribution  $p_{t+1}(x) = N(\hat{x}_{t+1}, \Sigma_{t+1})$  from the filtering distribution and (2.56)
5. Increment  $t$  by one and go to step 1

Repeating (2.57), the dynamics for  $\hat{x}_t$  and  $\Sigma_t$  are as follows

$$\begin{aligned}\hat{x}_{t+1} &= A\hat{x}_t + K_{\Sigma_t}(y_t - G\hat{x}_t) \\ \Sigma_{t+1} &= A\Sigma_tA' - K_{\Sigma_t}G\Sigma_tA' + Q\end{aligned}\tag{2.58}$$

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 (2.56) 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 (2.58):

$$\Sigma_{t+1} = A\Sigma_tA' - A\Sigma_tG'(G\Sigma_tG' + R)^{-1}G\Sigma_tA' + Q\tag{2.59}$$

This is a nonlinear difference equation in  $\Sigma_t$

A fixed point of (2.59) is a constant matrix  $\Sigma$  such that

$$\Sigma = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q\tag{2.60}$$

Equation (2.59) is known as a discrete time Riccati difference equation

Equation (2.60) 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 (2.59) converges to a nonnegative symmetric matrix  $\Sigma$  that solves (2.60)

### Implementation

The type `Kalman` from the `QuantEcon` package implements the Kalman filter

- Instance data:

- The parameters  $A, G, Q, R$  of a given model
- the moments  $(\hat{x}_t, \Sigma_t)$  of the current prior
- 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 (2.60) and the corresponding (stationary) Kalman gain

You can view the program [on GitHub](#) but we repeat it here for convenience

```
#=
Implements the Kalman filter for a linear Gaussian state space model.

@author : Spencer Lyon <spencer.lyon@nyu.edu>

@date: 2014-07-29

References
-----

http://quant-econ.net/jl/kalman.html

TODO: Do docstrings here after implementing LinerStateSpace
=#

type Kalman
    A
    G
    Q
    R
    k
    n
    cur_x_hat
    cur_sigma
end

# Initializes current mean and cov to zeros
function Kalman(A, G, Q, R)
    k = size(G, 1)
    n = size(G, 2)
    xhat = n == 1 ? zero(eltype(A)) : zeros(n)
    Sigma = n == 1 ? zero(eltype(A)) : zeros(n, n)
    return Kalman(A, G, Q, R, k, n, xhat, Sigma)
end
```

```

function set_state!(k::Kalman, x_hat, Sigma)
    k.cur_x_hat = x_hat
    k.cur_sigma = Sigma
    nothing
end

function prior_to_filtered!(k::Kalman, y)
    # simplify notation
    G, R = k.G, k.R
    x_hat, Sigma = k.cur_x_hat, k.cur_sigma

    # and then update
    if k.k > 1
        reshape(y, k.k, 1)
    end
    A = Sigma * G'
    B = G * Sigma' * G' + R
    M = A * inv(B)
    k.cur_x_hat = x_hat + M * (y - G * x_hat)
    k.cur_sigma = Sigma - M * G * Sigma
    nothing
end

function filtered_to_forecast!(k::Kalman)
    # simplify notation
    A, Q = k.A, k.Q
    x_hat, Sigma = k.cur_x_hat, k.cur_sigma

    # and then update
    k.cur_x_hat = A * x_hat
    k.cur_sigma = A * Sigma * A' + Q
    nothing
end

function update!(k::Kalman, y)
    prior_to_filtered!(k, y)
    filtered_to_forecast!(k)
    nothing
end

function stationary_values(k::Kalman)
    # simplify notation
    A, Q, G, R = k.A, k.Q, k.G, k.R

    # solve Riccati equation, obtain Kalman gain
    Sigma_inf = solve_discrete_riccati(A', G', Q, R)
    K_inf = A * Sigma_inf * G' * inv(G * Sigma_inf * G' + R)
    return Sigma_inf, K_inf
end

```

### 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 (2.56) with  $A = 1$ ,  $Q = 0$  and  $x_0 = \theta$

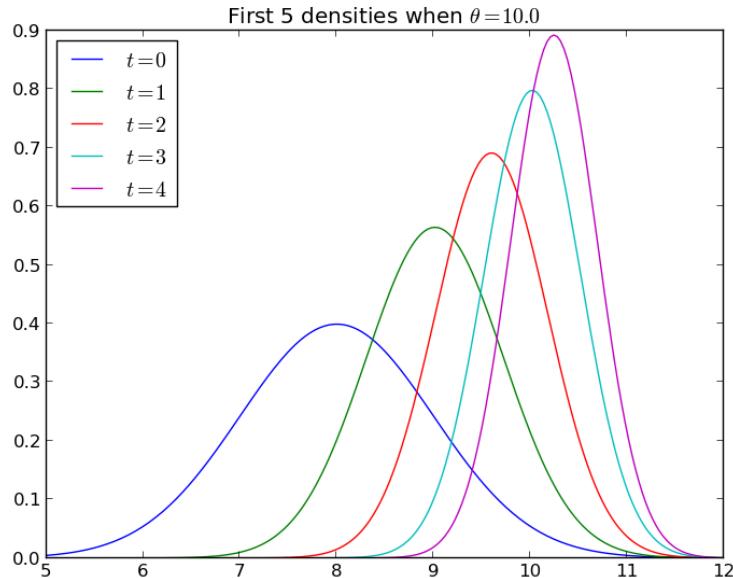
The measurement equation is  $y_t = \theta + v_t$  where  $v_t$  is  $N(0, 1)$  and iid

The task of this exercise is to simulate the model and, using the code from `kalman.jl`, 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  $\theta$

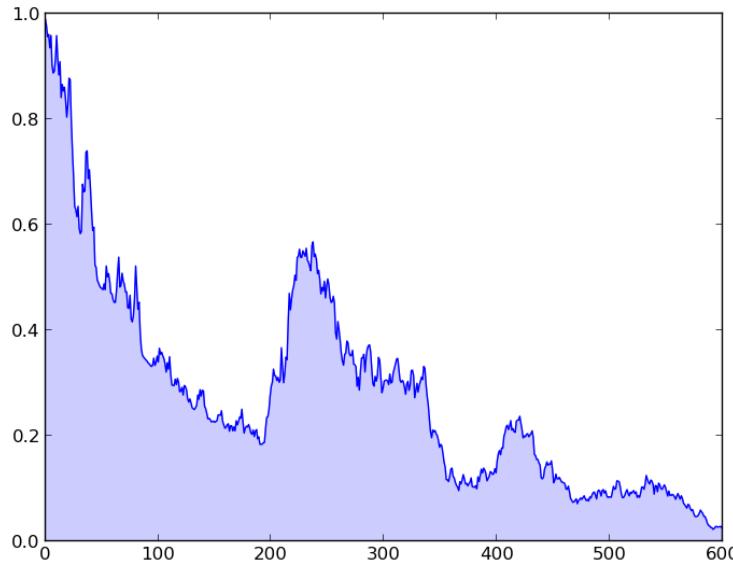
To get a better idea, choose a small  $\epsilon > 0$  and calculate

$$z_t := 1 - \int_{\theta-\epsilon}^{\theta+\epsilon} p_t(x) dx$$

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

$$A = \begin{pmatrix} 0.5 & 0.4 \\ 0.6 & 0.3 \end{pmatrix}$$

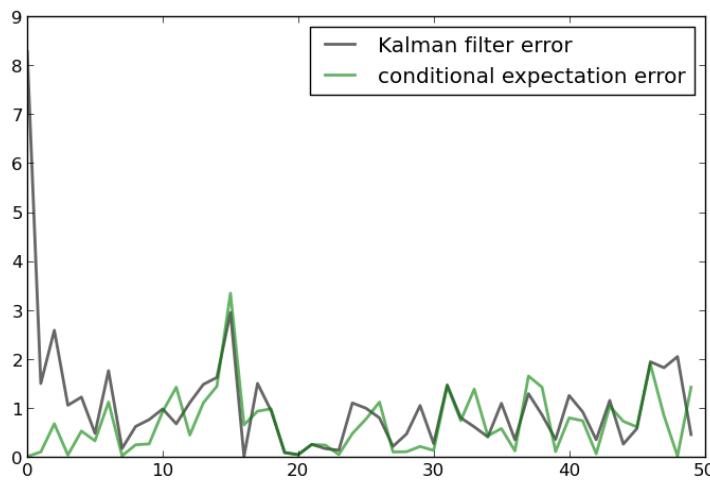
To initialize the prior density, set

$$\Sigma_0 = \begin{pmatrix} 0.9 & 0.3 \\ 0.3 & 0.9 \end{pmatrix}$$

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 (2.60)) 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](#)

## 2.8 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

$$x_m = \theta + \epsilon_m \quad \text{where} \quad \epsilon_m \sim N(0, \gamma_x^{-1}) \quad (2.61)$$

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  $\frac{1}{M} \sum_{m \in \mathbb{M}} x_m$  of the active firms

With this notation and primes for next period values, we can write the updating of the mean and precision via

$$\mu' = \rho \frac{\gamma \mu + M \gamma_x X}{\gamma + M \gamma_x} \quad (2.62)$$

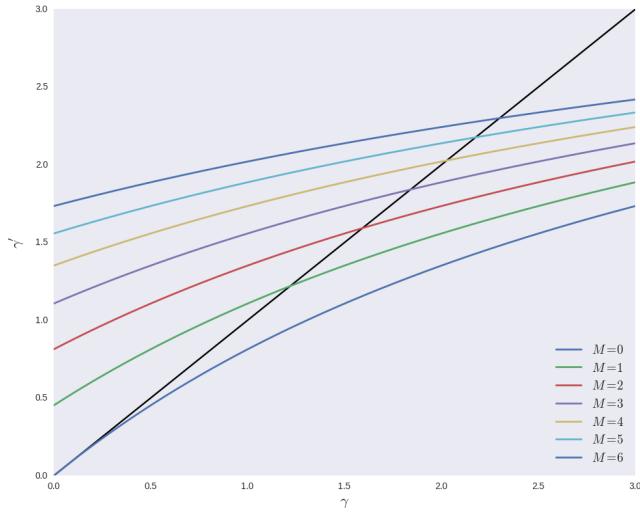
$$\gamma' = \left( \frac{\rho^2}{\gamma + M \gamma_x} + \sigma_\theta^2 \right)^{-1} \quad (2.63)$$

These are standard Kalman filtering results applied to the current setting

Exercise 1 provides more details on how (2.62) and (2.63) are derived, and then asks you to fill in remaining steps

The next figure plots the law of motion for the precision in (2.63) 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

$$\mathbb{E}[u(x_m - F_m)] > c \quad (2.64)$$

Here

- the mathematical expectation of  $x_m$  is based on (2.61) 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 (2.64)

The utility function has the constant absolute risk aversion form

$$u(x) = \frac{1}{a} (1 - \exp(-ax)) \quad (2.65)$$

where  $a$  is a positive parameter

Combining (2.64) and (2.65), 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

$$\psi(\mu, \gamma, F_m) := \frac{1}{a} \left( 1 - \exp \left( -a\mu + aF_m + \frac{a^2 \left( \frac{1}{\gamma} + \frac{1}{\gamma_x} \right)}{2} \right) \right) - c > 0 \quad (2.66)$$

### 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 (2.66) for each firm

The function *UncertaintyTrapEcon* encodes as default values the parameters we'll use in the simulations below

Here's the code, which can also be obtained from [GitHub](#)

```
type UncertaintyTrapEcon
    a::Float64          # Risk aversion
    gx::Float64          # Production shock precision
    rho::Float64          # Correlation coefficient for theta
    sig_theta::Float64    # Std dev of theta shock
    num_firms::Int        # Number of firms
    sig_F::Float64        # Std dev of fixed costs
    c::Float64            # External opportunity cost
    mu::Float64           # Initial value for mu
    gamma::Float64         # Initial value for gamma
    theta::Float64         # Initial value for theta
    sd_x::Float64          # standard deviation of shock
end

function UncertaintyTrapEcon(;a::Real=1.5, gx::Real=0.5, rho::Real=0.99,
                           sig_theta::Real=0.5, num_firms::Int=100,
                           sig_F::Real=1.5, c::Real=-420, mu_init::Real=0,
                           gamma_init::Real=4, theta_init::Real=0)
    sd_x = sqrt(a/gx)
    UncertaintyTrapEcon(a, gx, rho, sig_theta, num_firms, sig_F, c, mu_init,
                        gamma_init, theta_init, sd_x)
```

```

end

function psi(uc::UncertaintyTrapEcon, F)
    temp1 = -uc.a * (uc.mu - F)
    temp2 = 0.5 * uc.a^2 * (1/uc.gamma + 1/uc.gx)
    return (1/uc.a) * (1 - exp(temp1 + temp2)) - uc.c
end

"""
Update beliefs (mu, gamma) based on aggregates X and M.
"""

function update_beliefs!(uc::UncertaintyTrapEcon, X, M)
    # Simplify names
    gx, rho, sig_theta = uc.gx, uc.rho, uc.sig_theta

    # Update mu
    temp1 = rho * (uc.gamma*uc.mu + M*gx*X)
    temp2 = uc.gamma + M*gx
    uc.mu = temp1 / temp2

    # Update gamma
    uc.gamma = 1 / (rho^2 / (uc.gamma + M * gx) + sig_theta^2)
end

update_theta!(uc::UncertaintyTrapEcon, w) =
    (uc.theta = uc.rho*uc.theta + uc.sig_theta*w)

"""
Generate aggregates based on current beliefs (mu, gamma). This
is a simulation step that depends on the draws for F.
"""

function gen_aggregates(uc::UncertaintyTrapEcon)
    F_vals = uc.sig_F * randn(uc.num_firms)

    M = sum(psi(uc, F_vals) .> 0)::Int    # Counts number of active firms
    if M > 0
        x_vals = uc.theta + uc.sd_x * randn(M)
        X = mean(x_vals)
    else
        X = 0.0
    end
    return X, M
end

```

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 (2.62) and (2.63) 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

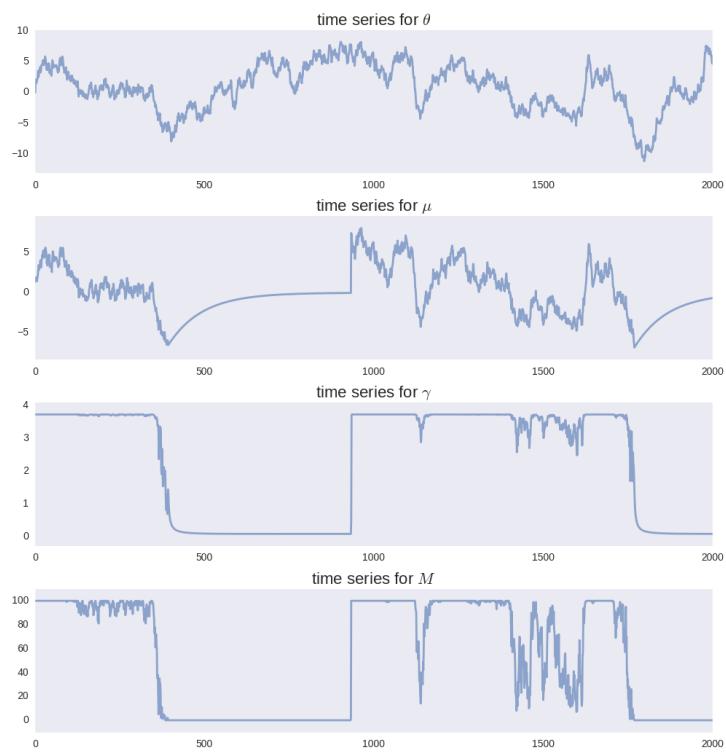
$$\pi(\theta | \mathbf{x}) = N(\mu_0, 1/\gamma_0)$$

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 function *UncertaintyTrapEcon*



## Solutions

[Solution notebook](#)

## 2.9 Infinite Horizon Dynamic Programming

### Contents

- *Infinite Horizon Dynamic Programming*
  - Overview
  - An Optimal Growth Model
  - Dynamic Programming
  - Computation
  - Writing Reusable Code
  - Exercises
  - Solutions

### Overview

In a [previous lecture](#) we gained some intuition about finite stage dynamic programming by studying the shortest path problem

The aim of this lecture is to introduce readers to methods for solving simple *infinite-horizon* dynamic programming problems using Julia

We will also introduce and motivate some of the modeling choices used throughout the lectures to treat this class of problems

The particular application we will focus on is solving for consumption in an optimal growth model

Although the model is quite specific, the key ideas extend to many other problems in dynamic optimization

The model is also very simplistic — we favor ease of exposition over realistic assumptions throughout the current lecture

**Other References** For supplementary reading see

- [\[LS12\]](#), section 3.1
- [EDTC](#), section 6.2 and chapter 10
- [\[Sun96\]](#), chapter 12
- [\[SLP89\]](#), chapters 2–5
- [\[HLL96\]](#), all

### An Optimal Growth 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:

$$k_{t+1} = y_t - c_t \quad (2.67)$$

Taking  $k_0$  as given, we suppose that the agent wishes to maximize

$$\sum_{t=0}^{\infty} \beta^t u(c_t) \quad (2.68)$$

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 (2.67) is always nonnegative
3. optimal in the sense that it maximizes (2.68) relative to all other feasible consumption sequences

A well-known result from the standard theory of dynamic programming (cf., e.g., [SLP89], section 4.1) states that, for kind of this problem, any optimal consumption sequence  $\{c_t\}$  must be *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

$$\sigma(k) \leq f(k) \quad \text{for all } k \in \mathbb{R}_+ \quad (2.69)$$

The set of all such policies will be denoted by  $\Sigma$

Using this notation, the agent's decision problem can be rewritten as

$$\max_{\sigma \in \Sigma} \left\{ \sum_{t=0}^{\infty} \beta^t u(\sigma(k_t)) \right\} \quad (2.70)$$

where the sequence  $\{k_t\}$  in (2.70) is given by

$$k_{t+1} = f(k_t) - \sigma(k_t), \quad k_0 \text{ given} \quad (2.71)$$

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

$$v_\sigma(k_0) := \sum_{t=0}^{\infty} \beta^t u(\sigma(k_t)) \quad (2.72)$$

when  $\{k_t\}$  is given by (2.71)

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

$$v^*(k_0) := \sup_{\sigma \in \Sigma} v_\sigma(k_0) \quad (2.73)$$

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 (2.73) for all  $k_0 \in \mathbb{R}_+$

The *Bellman equation* for this problem takes the form

$$v^*(k) = \max_{0 \leq c \leq f(k)} \{u(c) + \beta v^*(f(k) - c)\} \quad \text{for all } k \in \mathbb{R}_+ \quad (2.74)$$

It states that maximal value from a given state can be obtained by trading off current reward from a given action against 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

$$\max_{0 \leq c \leq f(k)} \{u(c) + \beta w(f(k) - c)\} \quad (2.75)$$

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 (2.74)

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: Starting with a guess — some initial function  $w$  — and successively improving it

The improvement step involves applying an “operator” (a mathematical term for a function that takes a function as an input and returns a new function as an output)

The operator in question is the *Bellman operator*

The Bellman operator for this problem is a map  $T$  sending function  $w$  into function  $Tw$  via

$$Tw(k) := \max_{0 \leq c \leq f(k)} \{u(c) + \beta w(f(k) - c)\} \quad (2.76)$$

Now let  $w$  be any continuous bounded function

It is known that iteratively applying  $T$  from initial condition  $w$  produces a sequence of functions  $w, Tw, T(Tw), \dots$  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 [\(2.76\)](#), 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 [\(2.76\)](#)

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 [\(2.76\)](#), 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

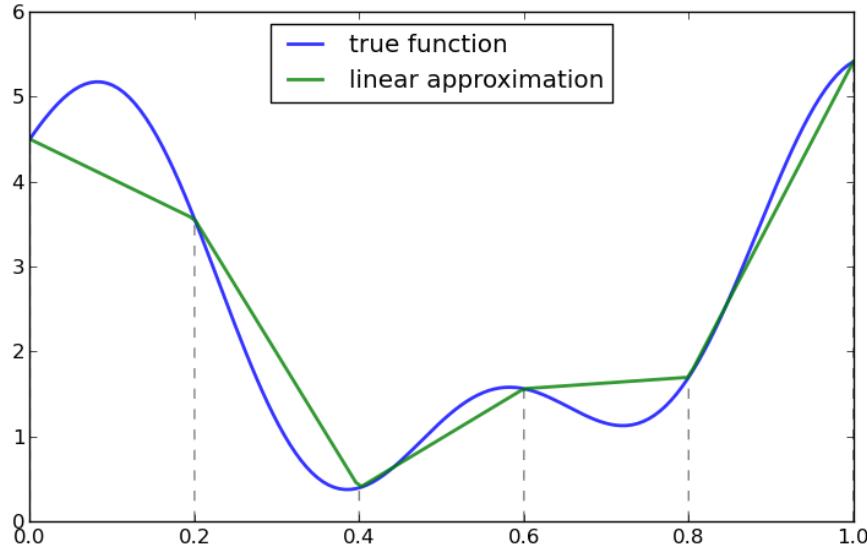
How should we go about step 2?

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 Julia

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

$$v^*(k) = c_1 + c_2 \ln k \quad (2.77)$$

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 `examples/optgrowth_v0.jl` from the [main repository](#)

We repeat it here for convenience

```

#=

A first pass at solving the optimal growth problem via value function
iteration. A more general version is provided in optgrowth.py.

@author : Spencer Lyon <spencer.lyon@nyu.edu>
=#

using Optim: optimize
using Grid: CoordInterpGrid, BCnan, InterpLinear
using PyPlot

## Primitives and grid
alpha = 0.65
bet = 0.95
grid_max = 2
grid_size = 150
grid = 1e-6:(grid_max-1e-6)/(grid_size-1):grid_max

## Exact solution
ab = alpha * bet
c1 = (log(1 - ab) + log(ab) * ab / (1 - ab)) / (1 - bet)
c2 = alpha / (1 - ab)
v_star(k) = c1 .+ c2 .* log(k)

function bellman_operator(grid, w)
    Aw = CoordInterpGrid(grid, w, BCnan, InterpLinear)

    Tw = zeros(w)

    for (i, k) in enumerate(grid)
        objective(c) = - log(c) - bet * Aw[k^alpha - c]
        res = optimize(objective, 1e-6, k^alpha)
        Tw[i] = - objective(res.minimum)
    end
    return Tw
end

function main(n::Int=35)
    w = 5 .* log(grid) .- 25 # An initial condition -- fairly arbitrary
    fig, ax = subplots()
    ax[:set_ylim](-40, -20)
    ax[:set_xlim](minimum(grid), maximum(grid))
    lb = "initial condition"
    jet = ColorMap("jet")[:__call__]
    ax[:plot](grid, w, color=jet(0), lw=2, alpha=0.6, label=lb)

    for i=1:n
        w = bellman_operator(grid, w)
        ax[:plot](grid, w, color=jet(i/n), lw=2, alpha=0.6)
    end
end

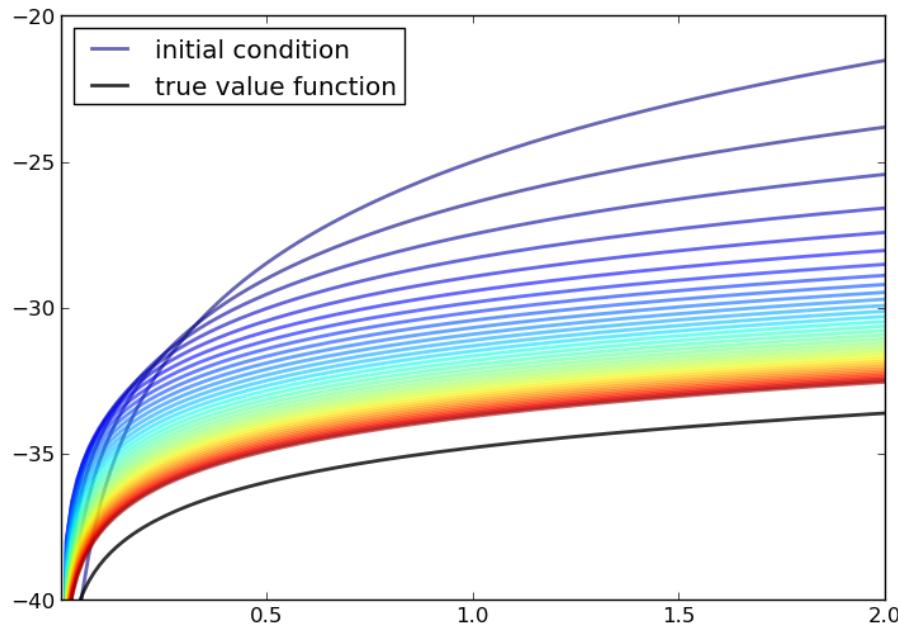
```

```

lb = "true value function"
ax[:plot](grid, v_star(grid), "k-", lw=2, alpha=0.8, label=lb)
ax[:legend](loc="upper left")
nothing
end

```

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 (2.77), drawn in black

The sequence of iterates converges towards  $v^*$

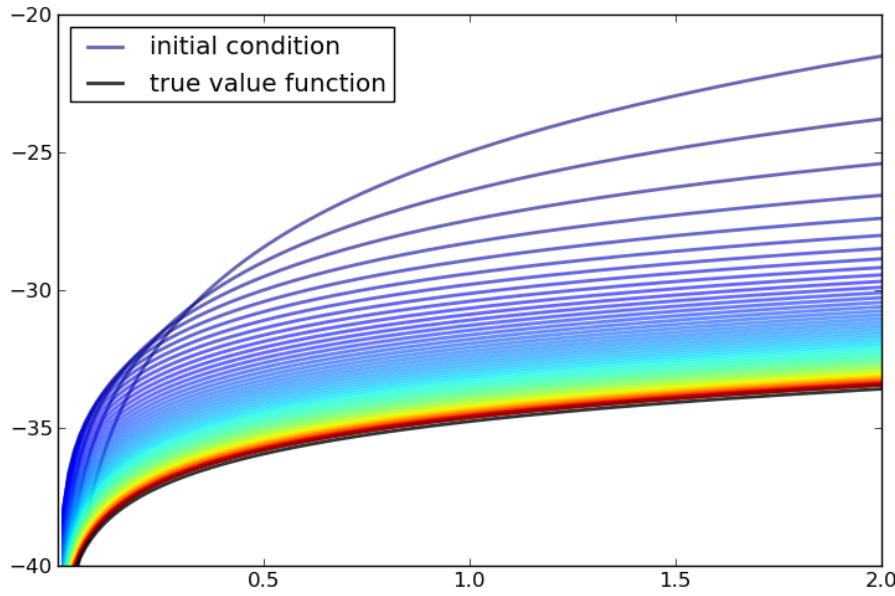
If we increase  $n$  and run again we see further improvement — the next figure shows  $n = 75$

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 the `getindex` method on the `CoordInterpGrid` from `Grid.jl`. The numerical solver `optimize` from `Optim.jl` minimizes its objective, so we use the identity  $\max_x f(x) = -\min_x -f(x)$  to solve (2.76)

Notice that we wrap the code used to generate the figure in a function named `main`. This allows us to import the functionality of `optgrowth_v0.jl` into a Julia session, 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

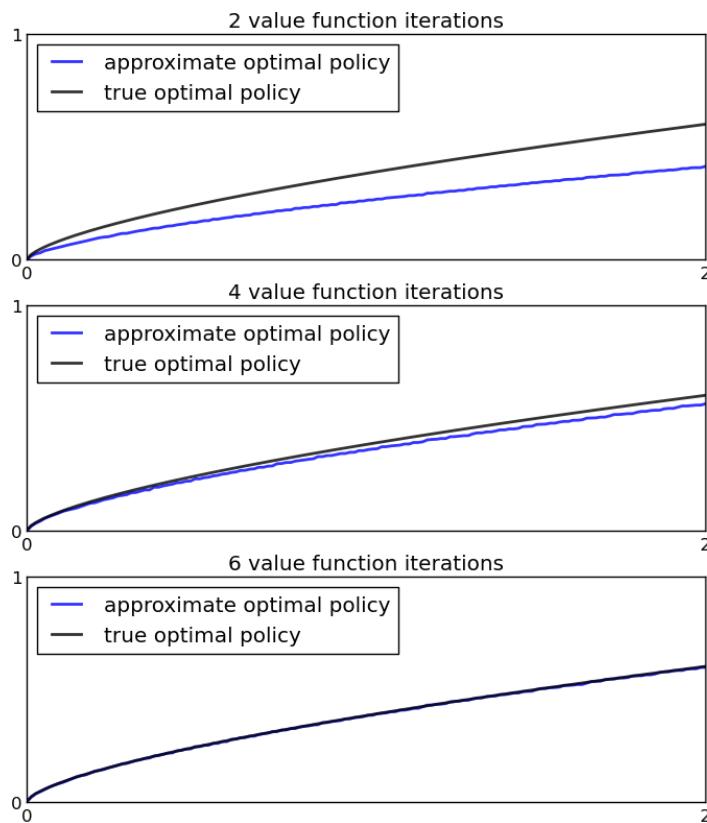
For this particular problem, the optimal consumption policy has the known analytical solution  $\sigma(k) = (1 - \alpha\beta)k^\alpha$

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 — although you should read the next section first



### Writing Reusable Code

The title of this section might sound uninteresting and a departure from our topic, but it's equally important if not more so

It's understandable that many economists never consider the basic principles of software development, preoccupied as they are with the applied aspects of trying to implement their projects

However, in programming as in many things, success tends to find those who focus on what is important, not just what is urgent

**The Danger of Copy and Paste** For computing the value function of the particular growth model studied above, the code we have already written (in file `optgrowth_v0.jl`, shown [here](#)) is perfectly fine

However, suppose that we now want to solve a different growth model, with different technology and preferences

Probably we want to keep our existing code, so let's follow our first instinct and

1. copy the contents of `optgrowth_v0.jl` to a new file
2. then make the necessary changes

Now let's suppose that we repeat this process again and again over several years, so we now have many similar files

(And perhaps we're doing similar things with other projects, leading to hundreds of specialized and partially related Julia files lying around our file system)

There are several potential problems here

**Problem 1** First, if we now realize we've been making some small but fundamental error with our dynamic programming all this time, we have to modify all these different files

And now we realize that we don't quite remember which files they were, and where exactly we put them...

So we fix all the ones we can find — spending a few hours in the process, since each implementation is slightly different and takes time to remember — and leave the ones we can't

Now, 6 weeks later, we need to use one of these files

But is file X one that we have fixed, or is it not?

In this way, our code base becomes a mess, with related functionality scattered across many files, and errors creeping into them

**Problem 2** A second issue here is that since all these files are specialized and might not be used again, there's little incentive to invest in writing them cleanly and efficiently

**DRY** The preceding discussion leads us to one of the most fundamental principles of code development: **don't repeat yourself**

To the extent that it's practical,

- always strive to write code that is abstract and generic in order to facilitate reuse
- try to ensure that each distinct logical concept is repeated in your code base as few times as possible

To this end, we are now going to rewrite our solution to the optimal growth problem given in `optgrowth_v0.jl` (shown [above](#)) with the intention of producing a more generic version

While some aspects of this exercise might seem like overkill, the principles are important, and easy to illustrate in the context of the current problem

**Implementation 2** In writing our second implementation, we want our function `bellman_operator` to be able to handle a wider class of models

In particular, we don't want model specifics hardwired into this function

Instead, we would like `bellman_operator` to act in conjunction with a more general description of a model (technology, preferences, etc.)

To do so it's convenient to wrap the model description up in a type and add the Bellman operator as a method

(Review [this lecture](#) if you have forgotten the syntax for type definitions)

This idea is implemented in the code below, in file `optgrowth.jl` from the `QuantEcon` package

```
#=
Solving the optimal growth problem via value function iteration.

@author : Spencer Lyon <spencer.lyon@nyu.edu>

@date : 2014-07-05

References
-----

Simple port of the file quantecon.models.optgrowth

http://quant-econ.net/jl/dp_intro.html
=#

#=

This type defines the primitives representing the growth model. The
default values are

    f(k) = k**alpha, i.e., Cobb-Douglas production function
    u(c) = ln(c), i.e., log utility

See the constructor below for details
=#

```

```

"""
Neoclassical growth model

##### Fields

- `f::Function` : Production function
- `bet::Real` : Discount factor in (0, 1)
- `u::Function` : Utility function
- `grid_max::Int` : Maximum for grid over savings values
- `grid_size::Int` : Number of points in grid for savings values
- `grid::FloatRange` : The grid for savings values

"""

type GrowthModel
    f::Function
    bet::Real
    u::Function
    grid_max::Int
    grid_size::Int
    grid::FloatRange
end

default_f(k) = k^0.65
default_u(c) = log(c)

"""

Constructor of `GrowthModel`


##### Arguments

- `f::Function(k->k^0.65)` : Production function
- `bet::Real(0.95)` : Discount factor in (0, 1)
- `u::Function(log)` : Utility function
- `grid_max::Int(2)` : Maximum for grid over savings values
- `grid_size::Int(150)` : Number of points in grid for savings values

"""

function GrowthModel(f=default_f, bet=0.95, u=default_u, grid_max=2,
                      grid_size=150)
    grid = linspace_range(1e-6, grid_max, grid_size)
    return GrowthModel(f, bet, u, grid_max, grid_size, grid)
end

"""

$(_-_bellman_main_docstring).

##### Arguments

- `g::GrowthModel` : Instance of `GrowthModel`
- `w::Vector` : Current guess for the value function
- `out::Vector` : Storage for output.
- `;ret_policy::Bool(false)` : Toggles return of value or policy functions

```

```

##### Returns

None, `out` is updated in place. If `ret_policy == true` out is filled with the
policy function, otherwise the value function is stored in `out`.

"""

function bellman_operator!(g::GrowthModel, w::Vector, out::Vector;
                           ret_policy::Bool=false)
    # Apply linear interpolation to w
    Aw = CoordInterpGrid(g.grid, w, BCnan, InterpLinear)

    for (i, k) in enumerate(g.grid)
        objective(c) = - g.u(c) - g.bet * Aw[g.f(k) - c]
        res = optimize(objective, 1e-6, g.f(k))
        c_star = res.minimum

        if ret_policy
            # set the policy equal to the optimal c
            out[i] = c_star
        else
            # set Tw[i] equal to max_c { u(c) + beta w(f(k_i) - c)}
            out[i] = - objective(c_star)
        end
    end

    return out
end

function bellman_operator(g::GrowthModel, w::Vector;
                           ret_policy::Bool=false)
    out = similar(w)
    bellman_operator!(g, w, out, ret_policy=ret_policy)
end

"""

$(_greedy_main_docstring).

#### Arguments

- `g::GrowthModel` : Instance of `GrowthModel`
- `w::Vector` : Current guess for the value function
- `out::Vector` : Storage for output

#### Returns

None, `out` is updated in place to hold the policy function

"""

function get_greedy!(g::GrowthModel, w::Vector, out::Vector)
    bellman_operator!(g, w, out, ret_policy=true)
end

get_greedy(g::GrowthModel, w::Vector) = bellman_operator(g, w, ret_policy=true)

```

Of course we could omit the type structure and just pass date to `bellman_operator` and `compute_greedy` as a list of separate arguments

For example

```
Tw = bellman_operator(f, beta, u, grid_max, grid_size, w)
```

This approach is also fine, and many prefer it

Our own view is that the type structure is more convenient and a bit less error prone because once an instance is created we can call the methods repeatedly without having to specify a lot of arguments

**Iteration** The next thing we need to do is implement iteration of the Bellman operator

Since iteratively applying an operator is something we'll do a lot of, let's write this as generic, reusable code

Our code is written in the file `compute_fp.jl` from the [main repository](#), and displayed below

```
#=
Compute the fixed point of a given operator T, starting from
specified initial condition v.

@author : Spencer Lyon <spencer.lyon@nyu.edu>

@date: 2014-07-05

References
-----

http://quant-econ.net/jl/dp_intro.html
=#

"""
Repeatedly apply a function to search for a fixed point

Approximates `T^∞ v`, where `T` is an operator (function) and `v` is an initial
guess for the fixed point. Will terminate either when `T^{k+1}(v) - T^k v <
err_tol` or `max_iter` iterations has been exceeded.

Provided that `T` is a contraction mapping or similar, the return value will
be an approximation to the fixed point of `T`.

##### Arguments

* `T`: A function representing the operator `T`
* `v::TV`: The initial condition. An object of type `TV`
* `;err_tol(1e-3)`: Stopping tolerance for iterations
* `;max_iter(50)`: Maximum number of iterations
* `;verbose(true)`: Whether or not to print status updates to the user
* `;print_skip(10)`: if `verbose` is true, how many iterations to apply between
print messages
```

```

##### Returns
---

* `::TV`: The fixed point of the operator `T`. Has type `TV`

##### Example

```julia
using QuantEcon
T(x, μ) = 4.0 * μ * x * (1.0 - x)
x_star = compute_fixed_point(x->T(x, 0.3), 0.4) # (4μ - 1)/(4μ)
```

"""

function compute_fixed_point{TV}(T::Function, v::TV; err_tol=1e-3,
                                 max_iter=50, verbose=true, print_skip=10)
    iterate = 0
    err = err_tol + 1
    while iterate < max_iter && err > err_tol
        new_v = T(v)::TV
        iterate += 1
        err = Base.maxabs(new_v - v)
        if verbose
            if iterate % print_skip == 0
                println("Compute iterate $iterate with error $err")
            end
        end
        v = new_v
    end

    if iterate < max_iter && verbose
        println("Converged in $iterate steps")
    elseif iterate == max_iter
        warn("max_iter exceeded in compute_fixed_point")
    end

    return v
end

```

As currently written, the code continues iteration until one of two stopping conditions holds

1. Successive iterates become sufficiently close together, in the sense that the maximum deviation between them falls below `error_tol`
2. The number of iterations exceeds `max_iter`

Examples of usage for all the code above can be found in the solutions to the exercises

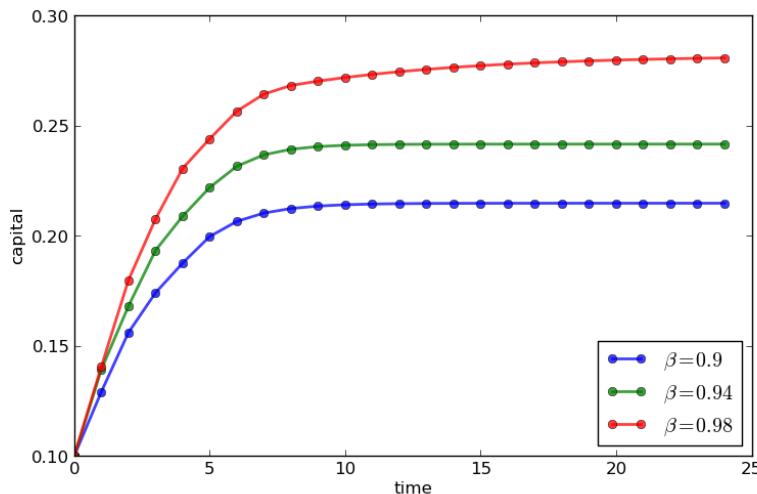
### Exercises

**Exercise 1** Replicate the optimal policy figure *shown above*

Use the same parameters and initial condition found in `optgrowth.jl`

**Exercise 2** Once an optimal consumption policy  $\sigma$  is given, the dynamics for the capital stock follows (2.71)

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.jl`

Replicate the figure

### Solutions

[Solution notebook](#)

## 2.10 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

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \quad t = 0, 1, 2, \dots \quad (2.78)$$

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 (2.78), 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

$$a_{t+1} = (1+r)a_t - u_t - \bar{c} + \sigma w_{t+1} + \mu \quad (2.79)$$

How can we write this new system in the form of equation (2.78)?

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

$$\begin{pmatrix} a_{t+1} \\ 1 \end{pmatrix} = \begin{pmatrix} 1+r & -\bar{c} + \mu \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_t \\ 1 \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix} u_t + \begin{pmatrix} \sigma \\ 0 \end{pmatrix} w_{t+1} \quad (2.80)$$

then the first row is equivalent to (2.79)

Moreover, the model is now linear, and can be written in the form of (2.78) by setting

$$x_t := \begin{pmatrix} a_t \\ 1 \end{pmatrix}, \quad A := \begin{pmatrix} 1+r & -\bar{c} + \mu \\ 0 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \end{pmatrix} \quad (2.81)$$

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

$$x_t' R x_t + u_t' Q u_t \quad (2.82)$$

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 (2.82) 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

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (x_t' R x_t + u_t' Q u_t) + \beta^T x_T' R_f x_T \right\} \quad (2.83)$$

subject to the law of motion (2.78) 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

$$J_{T-1}(x) := \min_u \{x' Rx + u' Qu + \beta \mathbb{E} J_T(Ax + Bu + Cw_T)\} \quad (2.84)$$

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, (2.84) becomes

$$J_{T-1}(x) := \min_u \{x' Rx + u' Qu + \beta \mathbb{E} (Ax + Bu + Cw_T)' P_T (Ax + Bu + Cw_T)\} \quad (2.85)$$

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

$$u = -(Q + \beta B' P_T B)^{-1} \beta B' P_T A x \quad (2.86)$$

Plugging this back into (2.85) and rearranging yields

$$J_{T-1}(x) := x' P_{T-1} x + d_{T-1}$$

where

$$P_{T-1} := R - \beta^2 A' P_T B (Q + \beta B' P_T B)^{-1} B' P_T A + \beta A' P_T A \quad (2.87)$$

and

$$d_{T-1} := \beta \operatorname{trace}(C' P_T C) \quad (2.88)$$

(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

$$P_{t-1} := R - \beta^2 A' P_t B (Q + \beta B' P_t B)^{-1} B' P_t A + \beta A' P_t A \quad \text{with} \quad P_T = R_f \quad (2.89)$$

and

$$d_{t-1} := \beta(d_t + \text{trace}(C'P_t C)) \quad \text{with} \quad d_T = 0 \quad (2.90)$$

Recalling (2.86), the minimizers from these backward steps are

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B' P_{t+1} B)^{-1} \beta B' P_{t+1} A \quad (2.91)$$

These are the linear optimal control policies we *discussed above*

In particular, the sequence of controls given by (2.91) and (2.78) solves our finite horizon LQ problem

Rephrasing this more precisely, the sequence  $u_0, \dots, u_{T-1}$  given by

$$u_t = -F_t x_t \quad \text{with} \quad x_{t+1} = (A - BF_t)x_t + Cw_{t+1} \quad (2.92)$$

for  $t = 0, \dots, T-1$  attains the minimum of (2.83) 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

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (c_t - \bar{c})^2 + \beta^T q a_T^2 \right\} \quad (2.93)$$

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 (2.79)

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 (2.81)

To match with this state and control, the objective function (2.93) can be written in the form of (2.83) by choosing

$$Q := 1, \quad R := \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad \text{and} \quad R_f := \begin{pmatrix} q & 0 \\ 0 & 0 \end{pmatrix}$$

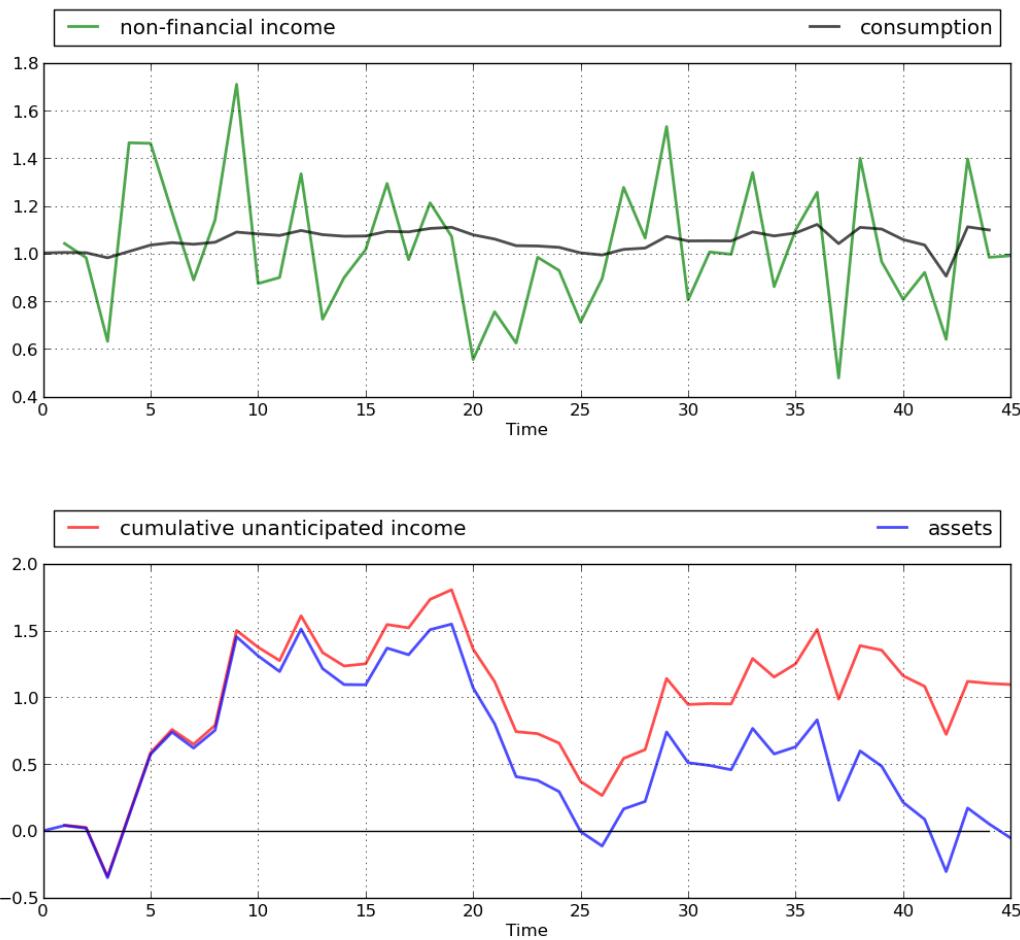
Now that the problem is expressed in LQ form, we can proceed to the solution by applying (2.89) and (2.91)

After generating shocks  $w_1, \dots, w_T$ , the dynamics for assets and consumption can be simulated via (2.92)

We provide code for all these operations below

The following figure was computed using this code, with  $r = 0.05, \beta = 1/(1+r), \bar{c} = 2, \mu = 1, \sigma = 0.25, T = 45$  and  $q = 10^6$

The shocks  $\{w_t\}$  were taken to be iid and standard normal



The top panel shows the time path of consumption  $c_t$  and income  $y_t$  in the simulation

As anticipated by the discussion on consumption smoothing, the time path of consumption is much smoother than that for income

(But note that consumption becomes more irregular towards the end of life, when the zero final asset requirement impinges more on consumption choices)

The second panel in the figure shows that the time path of assets  $a_t$  is closely correlated with cumulative unanticipated income, where the latter is defined as

$$z_t := \sum_{j=0}^t \sigma w_t$$

A key message is that unanticipated windfall gains are saved rather than consumed, while unanticipated negative shocks are met by reducing assets

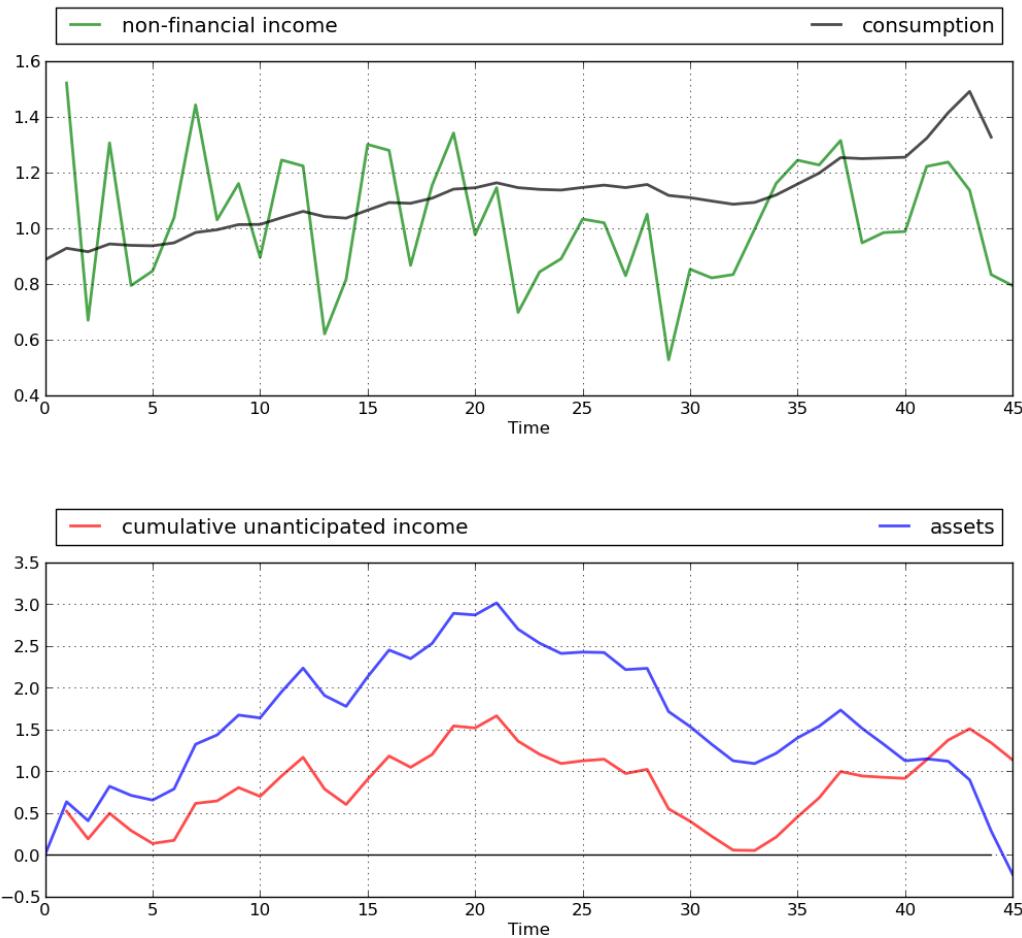
(Again, this relationship breaks down towards the end of life due to the zero final asset requirement)

These results are relatively robust to changes in parameters

For example, let's increase  $\beta$  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

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (x_t' R x_t + u_t' Q u_t + 2u_t' N x_t) + \beta^T x_T' R_f x_T \right\} \quad (2.94)$$

Our results extend to this case in a straightforward way

The sequence  $\{P_t\}$  from (2.89) becomes

$$P_{t-1} := R - (\beta B' P_t A + N)' (Q + \beta B' P_t B)^{-1} (\beta B' P_t A + N) + \beta A' P_t A \quad \text{with} \quad P_T = R_f \quad (2.95)$$

The policies in (2.91) are modified to

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B' P_{t+1} B)^{-1} (\beta B' P_{t+1} A + N) \quad (2.96)$$

The sequence  $\{d_t\}$  is unchanged from (2.90)

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

$$\mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t (x_t' R x_t + u_t' Q u_t + 2u_t' N x_t) \right\} \quad (2.97)$$

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 (2.89)

Equivalently, it is the solution  $P$  to the *discrete time algebraic Riccati equation*

$$P := R - (\beta B' P A + N)' (Q + \beta B' P B)^{-1} (\beta B' P A + N) + \beta A' P A \quad (2.98)$$

Equation (2.98) is also called the *LQ Bellman equation*, and the map that sends a given  $P$  into the right-hand side of (2.98) is called the *LQ Bellman operator*

The stationary optimal policy for this model is

$$u = -Fx \quad \text{where} \quad F := (Q + \beta B' P B)^{-1} (\beta B' P A + N) \quad (2.99)$$

The sequence  $\{d_t\}$  from (2.90) is replaced by the constant value

$$d := \text{trace}(C' P C) \frac{\beta}{1 - \beta} \quad (2.100)$$

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 (2.99) or (2.96)

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.jl` from the `QuantEcon` package

You can view the program on GitHub but we repeat it here for convenience

```
#=
Provides a type called LQ for solving linear quadratic control
problems.

@author : Spencer Lyon <spencer.lyon@nyu.edu>
@author : Zac Cranko <zaccranko@gmail.com>

@date : 2014-07-05

References
-----
http://quant-econ.net/jl/lqcontrol.html

=#
"""

Linear quadratic optimal control of either infinite or finite horizon

The infinite horizon problem can be written

    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

The finite horizon form is

    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.

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.

##### Fields

- `Q::ScalarOrArray` : k x k payoff coefficient for control variable u. Must be
symmetric and nonnegative definite
```

```

- `R::ScalarOrArray` : n x n payoff coefficient matrix for state variable x.
Must be symmetric and nonnegative definite
- `A::ScalarOrArray` : n x n coefficient on state in state transition
- `B::ScalarOrArray` : n x k coefficient on control in state transition
- `C::ScalarOrArray` : n x j coefficient on random shock in state transition
- `N::ScalarOrArray` : k x n cross product in payoff equation
- `bet::Real` : Discount factor in [0, 1]
- `capT::Union(Int, Nothing)` : Terminal period in finite horizon problem
- `rf::ScalarOrArray` : n x n terminal payoff in finite horizon problem. Must be
symmetric and nonnegative definite
- `P::ScalarOrArray` : n x n matrix in value function representation
V(x) = x'Px + d
- `d::Real` : Constant in value function representation
- `F::ScalarOrArray` : Policy rule that specifies optimal control in each period

"""
type LQ
    Q::ScalarOrArray
    R::ScalarOrArray
    A::ScalarOrArray
    B::ScalarOrArray
    C::ScalarOrArray
    N::ScalarOrArray
    bet::Real
    capT::Union(Int, Nothing) # terminal period
    rf::ScalarOrArray
    P::ScalarOrArray
    d::Real
    F::ScalarOrArray # policy rule
end

"""
Main constructor for LQ type

Specifies default arguments for all fields not part of the payoff function or
transition equation.

##### Arguments

- `Q::ScalarOrArray` : k x k payoff coefficient for control variable u. Must be
symmetric and nonnegative definite
- `R::ScalarOrArray` : n x n payoff coefficient matrix for state variable x.
Must be symmetric and nonnegative definite
- `A::ScalarOrArray` : n x n coefficient on state in state transition
- `B::ScalarOrArray` : n x k coefficient on control in state transition
- `C::ScalarOrArray(zeros(size(R, 1)))` : n x j coefficient on random shock in
state transition
- `N::ScalarOrArray(zeros(size(B, 1), size(A, 2)))` : k x n cross product in
payoff equation
- `bet::Real(1.0)` : Discount factor in [0, 1]
- `capT::Union(Int, Nothing)(nothing)` : Terminal period in finite horizon
problem
- `rf::ScalarOrArray(fill(NaN, size(R)...))` : n x n terminal payoff in finite

```

```

horizon problem. Must be symmetric and nonnegative definite.

"""

function LQ(Q::ScalarOrArray,
            R::ScalarOrArray,
            A::ScalarOrArray,
            B::ScalarOrArray,
            C::ScalarOrArray           = zeros(size(R, 1)),
            N::ScalarOrArray           = zero(B'A),
            bet::ScalarOrArray          = 1.0,
            capT::Union(Int, Nothing) = nothing,
            rf::ScalarOrArray           = fill(NaN, size(R)...))

    k = size(Q, 1)
    n = size(R, 1)
    F = k==n==1 ? zero(Float64) : zeros(Float64, k, n)
    P = copy(rf)
    d = 0.0

    LQ(Q, R, A, B, C, N, bet, capT, rf, P, d, F)
end

"""

Version of default constructor making `bet` `capT` `rf` keyword arguments

"""

function LQ(Q::ScalarOrArray,
            R::ScalarOrArray,
            A::ScalarOrArray,
            B::ScalarOrArray,
            C::ScalarOrArray           = zeros(size(R, 1)),
            N::ScalarOrArray           = zero(B'A);
            bet::ScalarOrArray          = 1.0,
            capT::Union(Int, Nothing) = nothing,
            rf::ScalarOrArray           = fill(NaN, size(R)...))
    LQ(Q, R, A, B, C, N, bet, capT, rf)
end

"""

Update `P` and `d` from the value function representation in finite horizon case

##### Arguments

- `lq::LQ` : instance of `LQ` type

##### Returns

- `P::ScalarOrArray` : n x n matrix in value function representation
   $V(x) = x'Px + d$ 
- `d::Real` : Constant in value function representation

##### Notes

```

This function updates the `P` and `d` fields on the `lq` instance in addition to returning them

```
"""
function update_values!(lq::LQ)
    # Simplify notation
    Q, R, A, B, N, C, P, d = lq.Q, lq.R, lq.A, lq.B, lq.N, lq.C, lq.P, lq.d

    # Some useful matrices
    s1 = Q + lq.bet * (B'P*B)
    s2 = lq.bet * (B'P*A) + N
    s3 = lq.bet * (A'P*A)

    # Compute F as (Q + B'PB)^{-1} (beta B'PA)
    lq.F = s1 \ s2

    # Shift P back in time one step
    new_P = R - s2'lq.F + s3

    # Recalling that trace(AB) = trace(BA)
    new_d = lq.bet * (d + trace(P * C * C'))

    # Set new state
    lq.P, lq.d = new_P, new_d
end

"""
Computes value and policy functions in infinite horizon model

##### Arguments

- `lq::LQ` : instance of `LQ` type

##### Returns

- `P::ScalarOrArray` : n x n matrix in value function representation
  V(x) = x'Px + d
- `d::Real` : Constant in value function representation
- `F::ScalarOrArray` : Policy rule that specifies optimal control in each period

##### Notes

This function updates the `P`, `d`, and `F` fields on the `lq` instance in addition to returning them

"""

function stationary_values!(lq::LQ)
    # simplify notation
    Q, R, A, B, N, C = lq.Q, lq.R, lq.A, lq.B, lq.N, lq.C

    # solve Riccati equation, obtain P
    A0, B0 = sqrt(lq.bet) * A, sqrt(lq.bet) * B
    P = solve_discrete_riccati(A0, B0, R, Q, N)
```

```

# Compute F
s1 = Q + lq.bet * (B' * P * B)
s2 = lq.bet * (B' * P * A) + N
F = s1 \ s2

# Compute d
d = lq.bet * trace(P * C * C') / (1 - lq.bet)

# Bind states
lq.P, lq.F, lq.d = P, F, d
end

"""
Non-mutating routine for solving for `P`, `d`, and `F` in infinite horizon model

See docstring for stationary_values! for more explanation
"""

function stationary_values(lq::LQ)
    _lq = LQ(copy(lq.Q),
              copy(lq.R),
              copy(lq.A),
              copy(lq.B),
              copy(lq.C),
              copy(lq.N),
              copy(lq.bet),
              lq.capT,
              copy(lq.rf))

    stationary_values!(_lq)
    return _lq.P, _lq.F, _lq.d
end

"""
Private method implementing `compute_sequence` when state is a scalar
"""

function _compute_sequence{T}(lq::LQ, x0::T, policies)
    capT = length(policies)

    x_path = Array(T, capT+1)
    u_path = Array(T, capT)

    x_path[1] = x0
    u_path[1] = -(first(policies)*x0)
    w_path = lq.C * randn(capT+1)

    for t = 2:capT
        f = policies[t]
        x_path[t] = lq.A*x_path[t-1] + lq.B*u_path[t-1] + w_path[t]
        u_path[t] = -(f*x_path[t])
    end
    x_path[end] = lq.A*x_path[capT] + lq.B*u_path[capT] + w_path[end]

    x_path, u_path, w_path

```

```

end

"""
Private method implementing `compute_sequence` when state is a scalar
"""

function _compute_sequence{T}(lq::LQ, x0::Vector{T}, policies)
    # Ensure correct dimensionality
    n, j, k = size(lq.C, 1), size(lq.C, 2), size(lq.B, 2)
    capT = length(policies)

    A, B, C = lq.A, reshape(lq.B, n, k), reshape(lq.C, n, j)

    x_path = Array(T, n, capT+1)
    u_path = Array(T, k, capT)
    w_path = [vec(C*randn(j)) for i=1:(capT+1)]

    x_path[:, 1] = x0
    u_path[:, 1] = -(first(policies)*x0)

    for t = 2:capT
        f = policies[t]
        x_path[:, t] = A*x_path[:, t-1] + B*u_path[:, t-1] + w_path[t]
        u_path[:, t] = -(f*x_path[:, t])
    end
    x_path[:, end] = A*x_path[:, capT] + B*u_path[:, capT] + w_path[end]

    x_path, u_path, w_path
end

"""

Compute and return the optimal state and control sequence, assuming w ~ N(0,1)

##### Arguments

- `lq::LQ` : instance of `LQ` type
- `x0::ScalarOrArray` : initial state
- `ts_length::Integer(100)` : maximum number of periods for which to return process. If `lq` instance is finite horizon type, the sequences are returned only for `min(ts_length, lq.capT)`

##### Returns

- `x_path::Matrix{Float64}` : An n x T+1 matrix, where the t-th column represents `x_t`
- `u_path::Matrix{Float64}` : A k x T matrix, where the t-th column represents `u_t`
- `w_path::Matrix{Float64}` : A j x T+1 matrix, where the t-th column represents `lq.C*w_t`

"""

function compute_sequence(lq::LQ, x0::ScalarOrArray, ts_length::Integer=100)
    capT = min(ts_length, lq.capT)

```

```

# Compute and record the sequence of policies
if isa(lq.capT, Nothing)
    stationary_values!(lq)
    policies = fill(lq.F, capT)
else
    policies = Array(typeof(lq.F), capT)
    for t = 1:capT
        update_values!(lq)
        policies[t] = lq.F
    end
end

_compute_sequence(lq, x0, policies)
end

```

In the module, the various updating, simulation and fixed point methods are wrapped in a type called `LQ`, which includes

- Instance data:
  - The required parameters  $Q, R, A, B$  and optional parameters  $C, \beta, T, R_f, N$  specifying a given LQ model
    - \* set  $T$  and  $R_f$  to `None` in the infinite horizon case
    - \* set  $C = \text{None}$  (or zero) in the deterministic case
  - the value function and policy data
    - \*  $d_t, P_t, F_t$  in the finite horizon case
    - \*  $d, P, F$  in the infinite horizon case
- Methods:
  - `update_values` — shifts  $d_t, P_t, F_t$  to their  $t - 1$  values via (2.89), (2.90) and (2.91)
  - `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.jl` from the [main 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)

```

sigma = 0.25
mu = 1.0
q = 1e6

# == Formulate as an LQ problem == #
Q = 1.0
R = zeros(2, 2)

```

```

Rf = zeros(2, 2); Rf[1, 1] = q
A = [1.0+r -c_bar+mu;
      0.0 1.0]
B = [-1.0, 0.0]
C = [sigma, 0.0]

# == Compute solutions and simulate ==
lq = LQ(Q, R, A, B, C; bet=bet, capT=T, rf=Rf)
x0 = [0.0, 1.0]
xp, up, wp = compute_sequence(lq, x0)

# == Convert back to assets, consumption and income ==
assets = squeeze(xp[1, :], 1)           # a_t
c = squeeze(up .+ c_bar, 1)              # c_t
income = squeeze(wp[1, 2:end] .+ mu, 1)   # y_t

# == Plot results ==
n_rows = 2
fig, axes = subplots(n_rows, 1, figsize=(12, 10))

subplots_adjust(hspace=0.5)
for i=1:n_rows
    axes[i][:grid]()
    axes[i][:set_xlabel]("Time")
end
bbox = [0.0, 1.02, 1.0, 0.102]

# Make first plot
axes[1][:plot](2:T+1, income, "g-", label="non-financial income", lw=2,
                alpha=0.7)
axes[1][:plot](1:T, c, "k-", label="consumption", lw=2, alpha=0.7)
axes[1][:legend](ncol=2, bbox_to_anchor=bbox, loc=3, mode="expand")

# Make second plot
axes[2][:plot](2:T+1, cumsum(income .- mu), "r-",
                label="cumulative unanticipated income", lw=2, alpha=0.7)
axes[2][:plot](1:T+1, assets, "b-", label="assets", lw=2, alpha=0.7)
axes[2][:plot](1:T, zeros(T), "k-")
axes[2][:legend](ncol=2, bbox_to_anchor=bbox, loc=3, mode="expand")

```

## 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

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (c_t - \bar{c})^2 + \beta^T q a_T^2 \right\} \quad (2.101)$$

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

$$a_{t+1} = (1+r)a_t - u_t - \bar{c} + m_1 t + m_2 t^2 + \sigma w_{t+1} \quad (2.102)$$

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

$$x_t := \begin{pmatrix} a_t \\ 1 \\ t \\ t^2 \end{pmatrix}, \quad A := \begin{pmatrix} 1+r & -\bar{c} & m_1 & m_2 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad (2.103)$$

If you expand the expression  $x_{t+1} = Ax_t + Bu_t + Cw_{t+1}$  using this specification, you will find that assets follow (2.102) as desired, and that the other state variables also update appropriately

To implement preference specification (2.101) we take

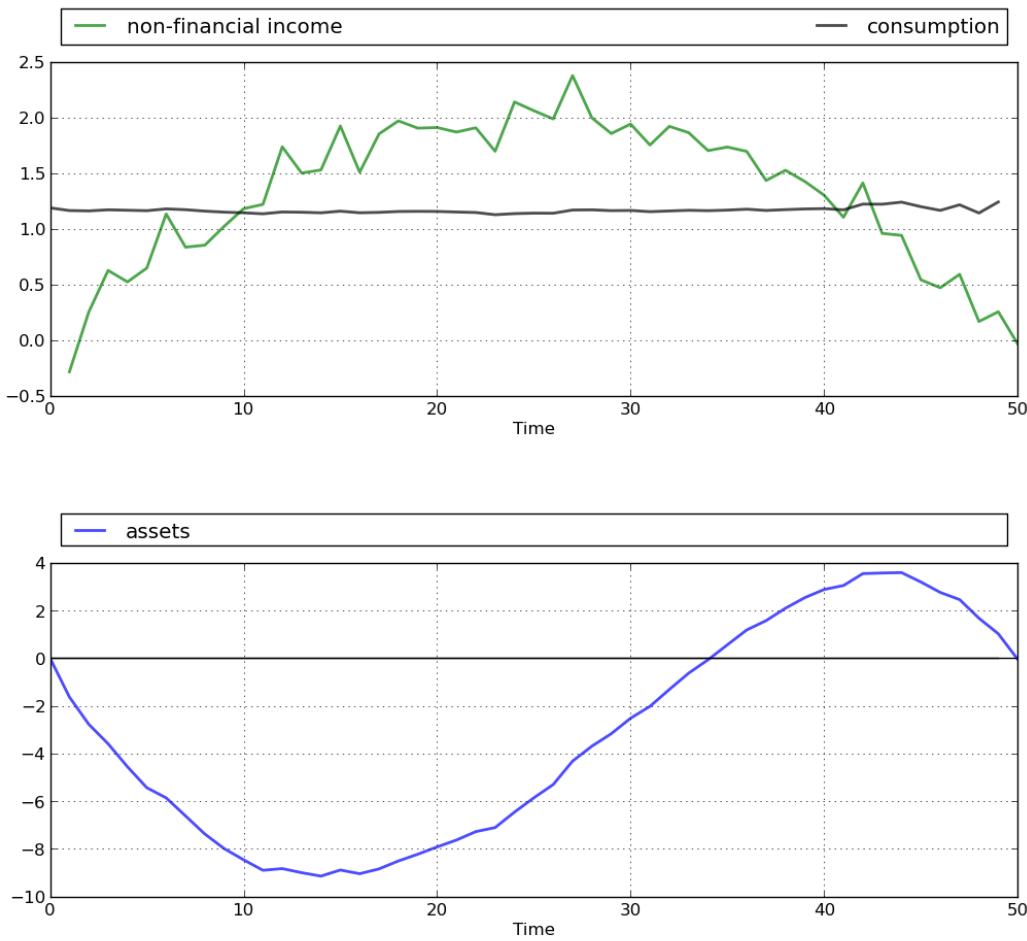
$$Q := 1, \quad R := \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad R_f := \begin{pmatrix} q & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad (2.104)$$

The next figure shows a simulation of consumption and assets computed using the `compute_sequence` method of `lqcontrol.jl` 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

$$y_t = \begin{cases} p(t) + \sigma w_{t+1} & \text{if } t \leq K \\ s & \text{otherwise} \end{cases} \quad (2.105)$$

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 (2.93)

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 (2.105) 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)

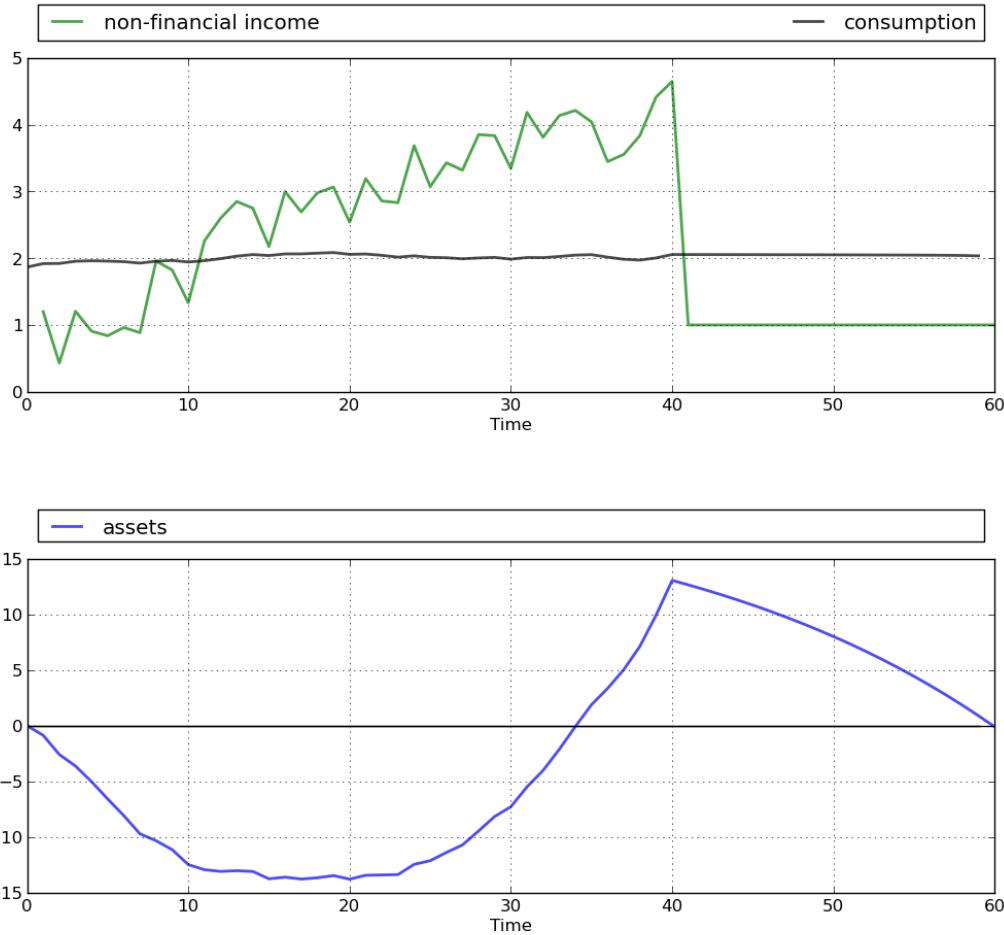
The basic idea is that although the whole problem is not a single time-invariant LQ problem, it is still a dynamic programming problem, and hence we can use appropriate Bellman equations at every stage

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

$$\mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t \pi_t \right\} \quad \text{where} \quad \pi_t := p_t q_t - c q_t - \gamma (q_{t+1} - q_t)^2 \quad (2.106)$$

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

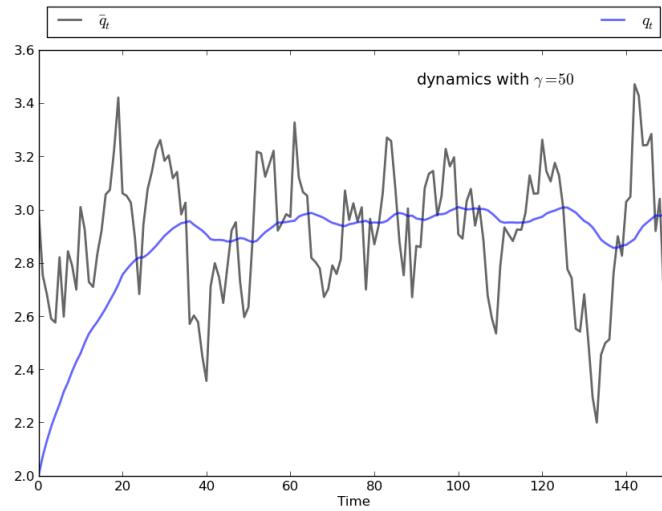
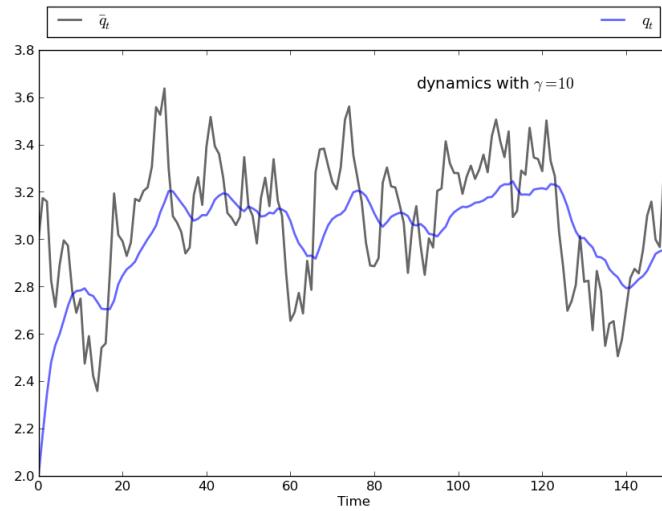
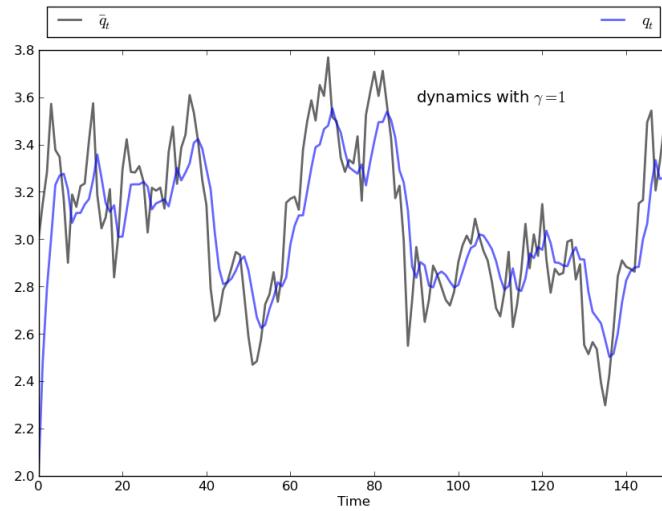
Here we take  $x_t = (\bar{q}_t \ q_t \ 1)'$ , while the control is chosen as  $u_t = q_{t+1} - q_t$

We also manipulated the profit function slightly

In (2.106), 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 (2.106) 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 (2.106), 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

$$\min \mathbb{E} \sum_{t=0}^{\infty} \beta^t \{ a_1(q_t - \bar{q}_t)^2 + \gamma u_t^2 \} \quad (2.107)$$

It's now relatively straightforward to find  $R$  and  $Q$  such that (2.107) can be written as (2.97)

Furthermore, the matrices  $A$ ,  $B$  and  $C$  from (2.78) 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 (2.103)

- 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 (2.104)

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](#)

## 2.11 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

$$p = a_0 - a_1 Y \quad (2.108)$$

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 (2.108), we can express the problem of the representative firm as

$$\max_y [(a_0 - a_1 Y)y - c_1 y - 0.5 c_2 y^2] \quad (2.109)$$

In posing problem (2.109), 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 (2.109)

This assures that the firm is a price taker

The first order condition for problem (2.109) is

$$a_0 - a_1 Y - c_1 - c_2 y = 0 \quad (2.110)$$

At this point, *but not before*, we substitute  $Y = ny$  into (2.110) to obtain the following linear equation

$$a_0 - c_1 - (a_1 + n^{-1}c_2)Y = 0 \quad (2.111)$$

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 (2.108)

**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

$$p_t = a_0 - a_1 Y_t \quad (2.112)$$

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

$$\sum_{t=0}^{\infty} \beta^t r_t \quad (2.113)$$

where

$$r_t := p_t y_t - \frac{\gamma(y_{t+1} - y_t)^2}{2}, \quad y_0 \text{ given} \quad (2.114)$$

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 (2.112), 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

$$Y_{t+1} = H(Y_t) \quad (2.115)$$

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 (2.115) 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

$$v(y, Y) = \max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \quad (2.116)$$

Let's denote the firm's optimal policy function by  $h$ , so that

$$y_{t+1} = h(y_t, Y_t) \quad (2.117)$$

where

$$h(y, Y) := \arg \max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \quad (2.118)$$

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

$$-\gamma(y' - y) + \beta v_y(y', H(Y)) = 0 \quad (2.119)$$

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 (2.116), giving

$$v_y(y, Y) = a_0 - a_1 Y + \gamma(y' - y)$$

Substituting this equation into (2.119) gives the *Euler equation*

$$-\gamma(y_{t+1} - y_t) + \beta[a_0 - a_1 Y_{t+1} + \gamma(y_{t+2} - y_{t+1})] = 0 \quad (2.120)$$

The firm optimally sets an output path that satisfies (2.120), taking (2.115) 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 (2.120) subject to the given initial condition  $y_0$  and the transversality condition

Note that solving the Bellman equation (2.116) for  $v$  and then  $h$  in (2.118) yields a decision rule that automatically imposes both the Euler equation (2.120) 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

$$Y_{t+1} = nh(Y_t/n, Y_t) \quad (2.121)$$

Thus, when firms believe that the law of motion for market-wide output is (2.115), their optimizing behavior makes the actual law of motion be (2.121)

**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 (2.115) and (2.121)

**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 (2.116)–(2.118), and a decision rule into an actual law via (2.121)

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>6</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])

---

<sup>6</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.

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  $t$

$$s(Y_t, Y_{t+1}) := \int_0^{Y_t} (a_0 - a_1 x) dx - \frac{\gamma(Y_{t+1} - Y_t)^2}{2} \quad (2.122)$$

The first term is the area under the demand curve, while the second measures the social costs of changing output

The *planning problem* is to choose a production plan  $\{Y_t\}$  to maximize

$$\sum_{t=0}^{\infty} \beta^t s(Y_t, Y_{t+1})$$

subject to an initial condition for  $Y_0$

**Solution of the Planning Problem** Evaluating the integral in (2.122) 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

$$V(Y) = \max_{Y'} \left\{ a_0 Y - \frac{a_1}{2} Y^2 - \frac{\gamma(Y' - Y)^2}{2} + \beta V(Y') \right\} \quad (2.123)$$

The associated first order condition is

$$-\gamma(Y' - Y) + \beta V'(Y') = 0 \quad (2.124)$$

Applying the same Benveniste-Scheinkman formula gives

$$V'(Y) = a_0 - a_1 Y + \gamma(Y' - Y)$$

Substituting this into equation (2.124) and rearranging leads to the Euler equation

$$\beta a_0 + \gamma Y_t - [\beta a_1 + \gamma(1 + \beta)] Y_{t+1} + \gamma \beta Y_{t+2} = 0 \quad (2.125)$$

**The Key Insight** Return to equation (2.120) 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 (2.125) and (2.120) 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 (2.123)

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

$$Y_{t+1} = \kappa_0 + \kappa_1 Y_t \quad (2.126)$$

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 (2.116) 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

$$y_{t+1} = h_0 + h_1 y_t + h_2 Y_t \quad (2.127)$$

Hence a rational expectations equilibrium will be defined by the parameters  $(\kappa_0, \kappa_1, h_0, h_1, h_2)$  in (2.126)–(2.127)

### Exercises

**Exercise 1** Consider the firm problem *described above*

Let the firm's belief function  $H$  be as given in (2.126)

Formulate the firm's problem as a discounted optimal linear regulator problem, being careful to describe all of the objects needed

Use the type LQ from the QuantEcon 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 (2.127) and give the values for each  $h_j$

If there were  $n$  identical competitive firms all behaving according to (2.127), what would (2.127) imply for the *actual* law of motion (2.115) 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 (2.126))

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 (2.112) 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](#)

## 2.12 Markov Asset Pricing

### Contents

- *Markov Asset Pricing*
  - *Overview*
  - *Pricing Models*
  - *Finite Markov Asset Pricing*
  - *Implementation*
  - *Exercises*
  - *Solutions*

“A little knowledge of geometric series goes a long way” – Robert E. Lucas, Jr.

### 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
- the pricing model, which determines how prices react to different income streams

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  $d_0, d_1, \dots$  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

$$p_t = d_t + \beta \mathbb{E}_t[p_{t+1}] \quad (2.128)$$

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 (buy today in exchange for the asset and dividend tomorrow), the basic risk-neutral asset pricing equation is

$$p_t = \beta \mathbb{E}_t[d_{t+1} + p_{t+1}] \quad (2.129)$$

**Pricing Under Risk Aversion** Let’s now introduce risk aversion by supposing that all agents evaluate payoffs according to strictly concave period utility function  $u$

In this setting Robert Lucas [Luc78] showed that under certain equilibrium conditions the price of an ex-dividend asset obeys the famous consumption-based asset pricing equation

$$p_t = \mathbb{E}_t \left[ \beta \frac{u'(d_{t+1})}{u'(d_t)} (d_{t+1} + p_{t+1}) \right] \quad (2.130)$$

Comparing (2.129) and (2.130), the difference is that  $\beta$  in (2.129) has been replaced by

$$\beta \frac{u'(d_{t+1})}{u'(d_t)}$$

This term is usually called the *stochastic discount factor*

We give a full derivation of (2.130) in a later lecture

For now we focus more on implications

For the most part we will assume preferences take the form

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \text{ with } \gamma > 0 \quad \text{or} \quad u(c) = \ln c$$

**Simple Examples** What price dynamics result from these models?

The answer to this question depends on the process we specify for 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 (2.128) 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

$$p_t = \frac{1}{1-\beta}d$$

This price is the equilibrium price in the constant dividend case

The ex-dividend equilibrium price is  $p_t = (1-\beta)^{-1}\beta d$

**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

$$p_t = \frac{d_t}{1-\beta\lambda} = \frac{\lambda^t d_0}{1-\beta\lambda} \quad (2.131)$$

(Hint: Set  $v_t = p_t/d_t$  in (2.128) and then  $v_t = v_{t+1} = v$  to solve for constant  $v$ )

The ex-dividend price is  $p_t = (1-\beta\lambda)^{-1}\beta\lambda d_t$

If, in this example, we take  $\lambda = 1 + g$ , 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 (2.131) 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

$$w = \beta(I - \beta\tilde{P})^{-1}Ps'$$

**Example 4: Deterministic dividends, risk averse pricing** Our formula for pricing a cum-dividend claim to a non random stream  $d_t = \lambda^t d$  then becomes

$$p_t = d_t + \beta\lambda^{-\gamma}p_{t+1}$$

Guessing again that the price obeys  $p_t = vd_t$  where  $v$  is a constant price-dividend ratio, we have  $vd_t = d_t + \beta\lambda^{-\gamma}vd_{t+1}$ , or

$$v = \frac{1}{1 - \beta\lambda^{1-\gamma}}$$

If  $u'(c) = 1/c$ , 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  $\lambda$

### Finite Markov Asset Pricing

For the remainder of the lecture we focus on computing asset prices when

- endowments follow a finite state Markov chain
- agents are risk averse, and prices obey (2.130)

Our finite state Markov setting emulates [MP85]

In particular, we'll assume that there is an endowment of a consumption good that follows

$$c_{t+1} = \lambda_{t+1}c_t \tag{2.132}$$

Here  $\lambda_t$  is governed by the  $n$  state Markov chain discussed *above*

A *Lucas tree* is a unit 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 (2.130), the definition of  $u$  and (2.132) leads to

$$p_t = \mathbb{E}_t \left[ \beta\lambda_{t+1}^{-\gamma}(c_{t+1} + p_{t+1}) \right] \tag{2.133}$$

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 (2.133) 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 (2.132) 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

$$v_i = \beta \sum_{j=1}^n P_{ij} s_j^{1-\gamma} (1 + v_j) \quad (2.134)$$

Letting  $\tilde{P}_{ij} = P_{ij} s_j^{1-\gamma}$ , we can write (2.134) 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

$$v = \beta(I - \beta \tilde{P})^{-1} \tilde{P} \mathbf{1} \quad (2.135)$$

With log preferences,  $\gamma = 1$  and hence  $s^{1-\gamma} = 1$ , from which we obtain

$$v = \frac{\beta}{1 - \beta} \mathbf{1}$$

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

$$p_t = \beta \mathbb{E}_t [\lambda_{t+1}^{-\gamma} (\zeta + p_{t+1})] \quad (2.136)$$

Now guess that the price takes the form

$$p_t = p(\lambda_t) = p_i \quad \text{when } \lambda_t = s_i$$

Then (2.136) 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

$$p = \beta(I - \beta \check{P})^{-1} \check{P} \zeta \mathbf{1} \quad (2.137)$$

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. to hold the option until next period

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 economy is identical with the one above

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

We can also write this as

$$w(s_i, p_S) = \max \left\{ \beta \sum_{j=1}^n P_{ij} s_j^{-\gamma} w(s_j, p_S), p(s_i) - p_S \right\} \quad (2.138)$$

Letting  $\hat{P}_{ij} = P_{ij} s_j^{-\gamma}$  and  $w_i = w(s_i, p_S)$ , we can express (2.138) as the nonlinear vector equation

$$w = \max \{ \beta \hat{P} w, p - p_S \mathbf{1} \} \quad (2.139)$$

To solve (2.139), 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 (2.138)

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 type `AssetPrices` from the `QuantEcon` package provides methods for computing some of the prices described above

We print the code here for convenience

```
#=
Computes asset prices in an endowment economy when the endowment obeys
geometric growth driven by a finite state Markov chain. The transition
matrix of the Markov chain is P, and the set of states is s. The
discount factor is beta, and gamma is the coefficient of relative risk
aversion in the household's utility function.

@author : Spencer Lyon <spencer.lyon@nyu.edu>

@date: 2014-06-27

References
-----
http://quant-econ.net/jl/markov_asset.html
=#

"""
A class to compute asset prices when the endowment follows a finite Markov chain

##### Fields

- `bet::Float64` : Discount factor in (0, 1)
- `P::Matrix{Float64}` : A valid stochastic matrix
- `s::Vector{Float64}` : Growth rate of consumption in each state
- `gamma::Float64` : Coefficient of risk aversion
- `n::Int(size(P, 1))` : The number of states
- `P_tilde::Matrix{Float64}` : modified transition matrix used in computing the
price of the lucas tree
- `P_check::Matrix{Float64}` : modified transition matrix used in computing the
price of the consol

"""
type AssetPrices
    bet::Real
    P::Matrix
    s::Vector
    gamm::Real
    n::Int
    P_tilde::Matrix
    P_check::Matrix
end

"""
Construct an instance of `AssetPrices`, where `n`, `P_tilde`, and `P_check` are
computed automatically for you. See also the documentation for the type itself
```

```

"""
function AssetPrices(bet::Real, P::Matrix, s::Vector, gamm::Real)
    P_tilde = P .* s'.^(1-gamm)
    P_check = P .* s'.^(-gamm)
    return AssetPrices(bet, P, s, gamm, size(P, 1), P_tilde, P_check)
end

"""
Computes the function v such that the price of the lucas tree is v(lambda)C_t

##### Arguments

- `ap::AssetPrices` : An instance of the `AssetPrices` type

##### Returns

- `v::Vector{Float64}` : the pricing function for the lucas tree

"""

function tree_price(ap::AssetPrices)
    # Simplify names
    P, s, gamm, bet, P_tilde = ap.P, ap.s, ap.gamm, ap.bet, ap.P_tilde

    # Compute v
    I = eye(ap.n)
    O = ones(ap.n)
    v = bet .* ((I - bet .* P_tilde)\ (P_tilde * O))
    return v
end

"""
Computes price of a consol bond with payoff zeta

##### Arguments

- `ap::AssetPrices` : An instance of the `AssetPrices` type
- `zeta::Float64` : Per period payoff of the consol

##### Returns

- `pbar::Vector{Float64}` : the pricing function for the lucas tree

"""

function consol_price(ap::AssetPrices, zet::Real)
    # Simplify names
    P, s, gamm, bet, P_check = ap.P, ap.s, ap.gamm, ap.bet, ap.P_check

    # Compute v
    I = eye(ap.n)
    O = ones(ap.n)
    v = bet .* ((I - bet .* P_check)\ (P_check * (zet .* O)))
    return v
end

```

```

"""
Computes price of a call option on a consol bond, both finite and infinite
horizon

##### Arguments

- `zeta::Float64` : Coupon of the console
- `p_s::Float64` : Strike price
- `T::Vector{Int}(Int[])` : Time periods for which to store the price in the
finite horizon version
- `epsilon::Float64` : Tolerance for infinite horizon problem

##### Returns

- `w_bar::Vector{Float64}` Infinite horizon call option prices
- `w_bars::Dict{Int, Vector{Float64}}` 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

"""

function call_option(ap::AssetPrices, zet::Real, p_s::Real,
                    T::Vector{Int}=Int[], epsilon=1e-8)
    # Simplify names, initialize variables
    P, s, gamm, bet, P_check = ap.P, ap.s, ap.gamm, ap.bet, ap.P_check

    # Compute consol price
    v_bar = consol_price(ap, zet)

    # Compute option price
    w_bar = zeros(ap.n)

    err = epsilon + 1.0
    t = 0
    w_bars = Dict{Int, Vector{eltype(P)}}()
    while err > epsilon
        if t in T w_bars[t] = w_bar end
        # Maximize across columns
        w_bar_new = max(bet .* (P_check * w_bar), v_bar.-p_s)
        # Find maximal difference of each component
        err = Base.maxabs(w_bar - w_bar_new)
        # Update
        w_bar = w_bar_new
        t += 1
    end
    return w_bar, w_bars
end

```

## Exercises

**Exercise 1** Compute the price of the Lucas tree in an economy with the following primitives

```

n = 5
P = 0.0125 .* ones(n, n)
P .+= diagm(0.95 - 0.0125 .* ones(5))
s = [1.05, 1.025, 1.0, 0.975, 0.95]
gamm = 2.0
bet = 0.94
zet = 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](#)

## 2.13 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

$$\mathbb{E}_0 \left[ \sum_{t=0}^{\infty} \beta^t u(c_t) \right] \tag{2.140}$$

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

$$b_{t+1} = (1 + r)(c_t + b_t - y_t) \quad t \geq 0 \quad (2.141)$$

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

$$x_{t+1} = Ax_t + Cw_{t+1} \quad (2.142)$$

$$y_t = Ux_t \quad (2.143)$$

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

$$\mathbb{E}_0 \left[ \sum_{t=0}^{\infty} \beta^t b_t^2 \right] < \infty \quad (2.144)$$

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 (2.140) subject to (2.141) are

$$\mathbb{E}_t[u'(c_{t+1})] = u'(c_t), \quad t = 0, 1, \dots \quad (2.145)$$

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, (2.145) has the striking implication that consumption follows a martingale:

$$\mathbb{E}_t[c_{t+1}] = c_t \quad (2.146)$$

(In fact quadratic preferences are *necessary* for this conclusion <sup>7</sup>)

One way to interpret (2.146) 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>8</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

---

We want to solve the system of difference equations formed by (2.141) and (2.146) subject to the boundary condition (2.144)

To accomplish this, observe first that (2.144) implies  $\lim_{t \rightarrow \infty} \beta^t b_{t+1} = 0$

Using this restriction on the debt path and solving (2.141) forward yields

$$b_t = \sum_{j=0}^{\infty} \beta^j (y_{t+j} - c_{t+j}) \quad (2.147)$$

Take conditional expectations on both sides of (2.147) and use the *law of iterated expectations* to deduce

$$b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - \frac{c_t}{1 - \beta} \quad (2.148)$$

---

<sup>7</sup> A linear marginal utility is essential for deriving (2.146) from (2.145). 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 (2.145). 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 (2.145) 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>8</sup> An optimal decision rule is a map from current state into current actions—in this case, consumption

Expressed in terms of  $c_t$  we get

$$c_t = (1 - \beta) \left[ \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - b_t \right] \quad (2.149)$$

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 (2.149) 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:

$$x_{t+1} = Ax_t + Cw_{t+1} \quad (2.150)$$

$$b_{t+1} = b_t + U[(I - \beta A)^{-1}(A - I)]x_t \quad (2.151)$$

$$y_t = Ux_t \quad (2.152)$$

$$c_t = (1 - \beta)[U(I - \beta A)^{-1}x_t - b_t] \quad (2.153)$$

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 (2.150) as

$$z_{t+1} = \tilde{A}z_t + \tilde{C}w_{t+1} \quad (2.154)$$

$$\tilde{y}_t = \tilde{U}z_t \quad (2.155)$$

We can use the following formulas from [state-space representation](#) to compute population mean  $\mu_t = \mathbb{E} z_t$  and covariance  $\Sigma_t := \mathbb{E} [(z_t - \mu_t)(z_t - \mu_t)']$

$$\mu_{t+1} = \tilde{A}\mu_t \quad \text{with } \mu_0 \text{ given} \quad (2.156)$$

$$\Sigma_{t+1} = \tilde{A}\Sigma_t\tilde{A}' + \tilde{C}\tilde{C}' \quad \text{with } \Sigma_0 \text{ given} \quad (2.157)$$

We can then compute the mean and covariance of  $\tilde{y}_t$  from

$$\mu_{y,t} = \tilde{U}\mu_t\Sigma_{y,t} = \tilde{U}\Sigma_t\tilde{U}' \quad (2.158)$$

**A Simple Example with iid Income** To gain some preliminary intuition on the implications of (2.150), 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

$$b_t = -\sigma \sum_{j=1}^{t-1} w_j$$

$$c_t = \mu + (1-\beta)\sigma \sum_{j=1}^t w_j$$

Thus income is iid and debt and consumption are both Gaussian random walks

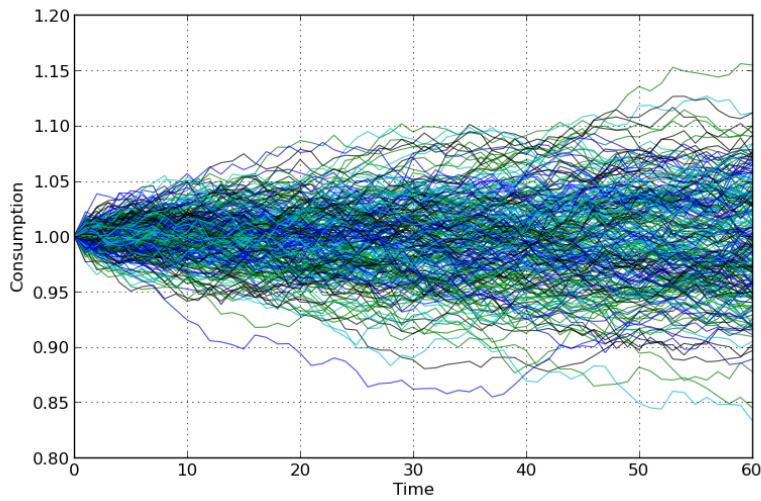
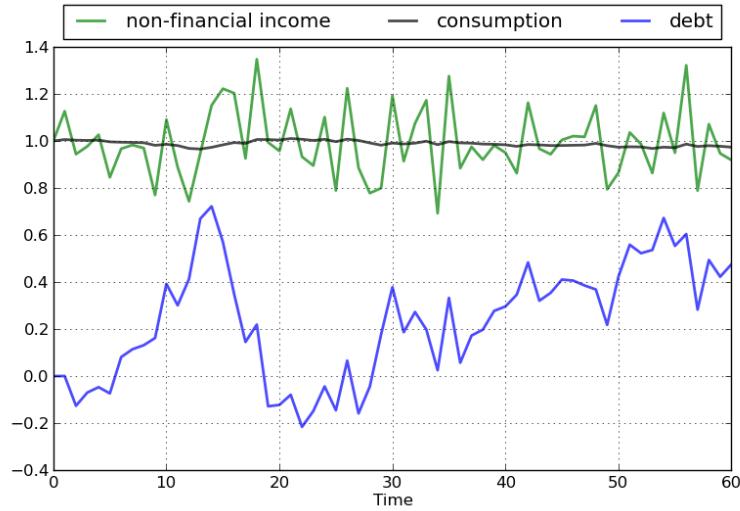
Defining assets as  $-b_t$ , we see that assets are just the cumulative sum of unanticipated income prior to the present date

The next figure shows a typical realization with  $r = 0.05$ ,  $\mu = 1$  and  $\sigma = 0.15$

Observe that consumption is considerably smoother than income

The figure below shows the consumption paths of 250 consumers with independent income streams

The code for these figures can be found in [perm\\_inc\\_figs.jl](#)



### 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 (2.149) forward one period and eliminate  $b_{t+1}$  by using (2.141) 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

$$c_{t+1} - c_t = (1 - \beta) \sum_{j=0}^{\infty} \beta^j \{ \mathbb{E}_{t+1}[y_{t+j+1}] - \mathbb{E}_t[y_{t+j+1}] \} \quad (2.159)$$

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 (2.159) and (2.148), which is repeated here:

$$b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - \frac{1}{1 - \beta} c_t \quad (2.160)$$

Equation (2.160) 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 (2.142) and substituting into (2.159) and (2.160) gives the following representation for the consumer's optimum decision rule:

$$c_{t+1} = c_t + (1 - \beta) U(I - \beta A)^{-1} C w_{t+1} \quad (2.161)$$

$$b_t = U(I - \beta A)^{-1} x_t - \frac{1}{1 - \beta} c_t \quad (2.162)$$

$$y_t = U x_t \quad (2.163)$$

$$x_{t+1} = A x_t + C w_{t+1} \quad (2.164)$$

Representation (2.161) 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 (2.146)

**Cointegration** Representation (2.161) 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>9</sup>

Despite this, both  $c_t$  and  $b_t$  will be non-stationary because they have unit roots (see (2.150) 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 (2.161) we have

$$(1 - \beta)b_t + c_t = (1 - \beta)U(I - \beta A)^{-1}x_t \quad (2.165)$$

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 (2.165) can be arranged to take the form

$$(1 - \beta)b_t + c_t = (1 - \beta)\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j}, \quad (2.166)$$

Equation (2.166) asserts that the *cointegrating residual* on the left side equals the conditional expectation of the geometric sum of future incomes on the right<sup>10</sup>

**Cross-Sectional Implications** Consider again (2.161), this time in light of our discussion of distribution dynamics in the [lecture on linear systems](#)

The dynamics of  $c_t$  are given by

$$c_{t+1} = c_t + (1 - \beta)U(I - \beta A)^{-1}Cw_{t+1} \quad (2.167)$$

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>9</sup> This would be the case if, for example, the *spectral radius* of  $A$  is strictly less than one

<sup>10</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

$$\text{Var}[c_t] = \text{Var}[c_0] + t \hat{\sigma}^2 \quad (2.168)$$

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 (2.168) 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 (2.142) has the moving average representation

$$y_{t+1} = d(L)w_{t+1} \quad (2.169)$$

where

- $d(L) = \sum_{j=0}^{\infty} d_j L^j$  for some sequence  $d_j$ , where  $L$  is the lag operator <sup>11</sup>
- at time  $t$ , the household has an information set <sup>12</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

$$\mathbb{E}_{t+1}[y_{t+j}] - \mathbb{E}_t[y_{t+j}] = d_{j-1} w_{t+1} \quad (2.170)$$

Using (2.170) in (2.159) gives

$$c_{t+1} - c_t = (1 - \beta)d(\beta)w_{t+1} \quad (2.171)$$

The object  $d(\beta)$  is the *present value of the moving average coefficients* in the representation for the endowment process  $y_t$

<sup>11</sup> Representation (2.142) implies that  $d(L) = U(I - \beta A)^{-1}C$ .

<sup>12</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 (2.161) we have

$$c_{t+1} - c_t = \sigma_1 w_{1t+1} + (1 - \beta) \sigma_2 w_{2t+1} \quad (2.172)$$

Formula (2.172) 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 (2.150) to this example shows that

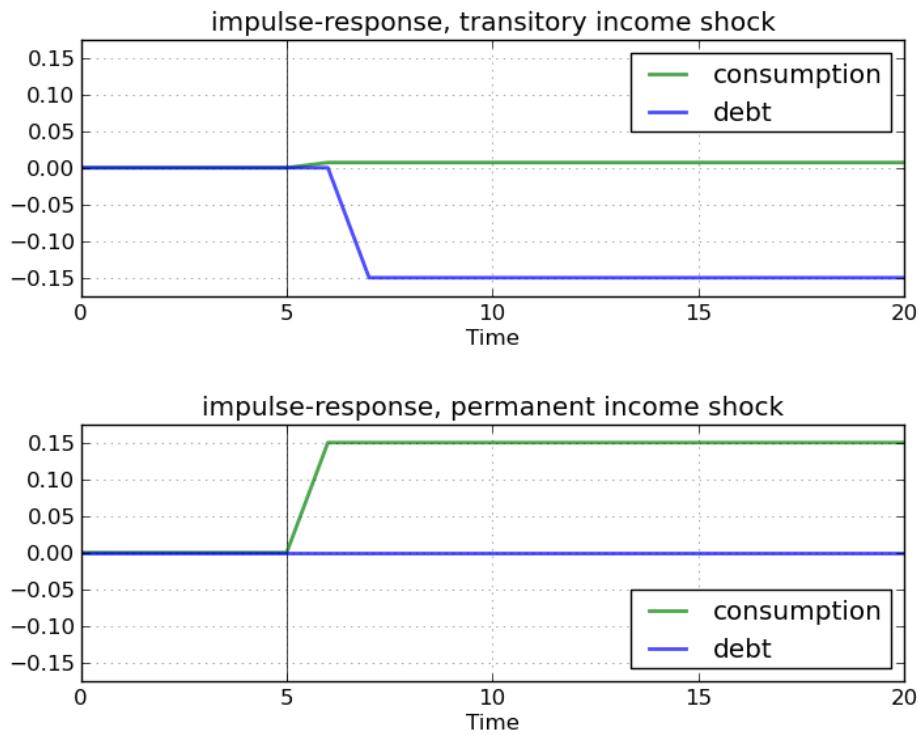
$$b_{t+1} - b_t = -x_{2t} = -\sigma_2 w_{2t} \quad (2.173)$$

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 `examples/perm_inc_ir.jl` from the [main repository](#), as shown below

```
#=
@author : Spencer Lyon
@date: 07/09/2014
=#
using PyPlot
```



```

const r = 0.05
const beta = 1.0 / (1.0 + r)
const T = 20 # Time horizon
const S = 5 # Impulse date
const sigma1 = 0.15
const sigma2 = 0.15

function time_path(permanent=false)
    w1 = zeros(T+1)
    w2 = zeros(T+1)
    b = zeros(T+1)
    c = zeros(T+1)

    if permanent === false
        w2[S+2] = 1.0
    else
        w1[S+2] = 1.0
    end

    for t=2:T
        b[t+1] = b[t] - sigma2 * w2[t]
        c[t+1] = c[t] + sigma1 * w1[t+1] + (1 - beta) * sigma2 * w2[t+1]
    end

    return b, c

```

```

end

function main()
    fix, axes = 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]("Time")
        ax[:set_ylim](-L, L)
        ax[:plot]((S, S), (-L, L), "k-", lw=0.5)
    end

    ax = axes[1]
    b, c = time_path(false)
    ax[:set_title]("impulse-response, transitory income shock")
    ax[:plot](0:T, c, "g-", label="consumption"; p_args...)
    ax[:plot](0:T, b, "b-", label="debt"; p_args...)
    ax[:legend](loc="upper right")

    ax = axes[2]
    b, c = time_path(true)
    ax[:set_title]("impulse-response, permanent income shock")
    ax[:plot](0:T, c, "g-", label="consumption"; p_args...)
    ax[:plot](0:T, b, "b-", label="debt"; p_args...)
    ax[:legend](loc="lower right")

    return nothing
end

```

**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 (2.161)

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 (2.161) implies

$$c_{t+1} - c_t = [1 - \beta(1 - K)]a_{t+1} \quad (2.174)$$

where the endowment process can now be represented in terms of the univariate innovation to  $y_t$  as

$$y_{t+1} - y_t = a_{t+1} - (1 - K)a_t \quad (2.175)$$

Equation (2.175) 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 (2.175), the first difference of income is a first-order moving average

Equation (2.174) asserts that the first difference of consumption is iid

Application of formula to this example shows that

$$b_{t+1} - b_t = (K - 1)a_t \quad (2.176)$$

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 (2.145) 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 (2.141) 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 (2.145) in the two period case

The proof for the general case is not dissimilar



## ADVANCED APPLICATIONS

This advanced section of the course contains more complex applications, and can be read selectively, according to your interests

### 3.1 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 Julia 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 the `kde` function from [KernelDensity.jl](#)

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 type called LAE for estimating densities by this technique can be found in QuantEcon

We repeat it here for convenience

```
#=
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 X_t is

1/n sum_{i=0}^n p(X_{t-1})^i, y)

This is a density in y.

@author : Spencer Lyon <spencer.lyon@nyu.edu>
@date: 2014-08-01

References
-----
http://quant-econ.net/jl/stationary_densities.html
=#
"""

A look ahead estimator associated with a given stochastic kernel p and a vector
of observations X.

##### Fields
```

```

- `p::Function`: The stochastic kernel. Signature is `p(x, y)` and it should be
vectorized in both inputs
- `X::Matrix`: A vector containing observations. Note that this can be passed as
any kind of `AbstractArray` and will be coerced into an `n x 1` vector.

"""
type LAE
    p::Function
    X::Matrix

    function LAE(p::Function, X::AbstractArray)
        n = length(X)
        new(p, reshape(X, n, 1))
    end
end

"""
A vectorized function that returns the value of the look ahead estimate at the
values in the array y.

##### Arguments

- `l::LAE`: Instance of `LAE` type
- `y::Array`: Array that becomes the `y` in `l.p(l.x, y)`

##### Returns

- `psi_vals::Vector`: Density at `(x, y)`

"""

function lae_est{T}(l::LAE, y::AbstractArray{T})
    k = length(y)
    v = l.p(l.X, reshape(y, 1, k))
    psi_vals = mean(v, 1)
    return squeeze(psi_vals, 1)
end

```

This function returns the right-hand side of (3.13) using

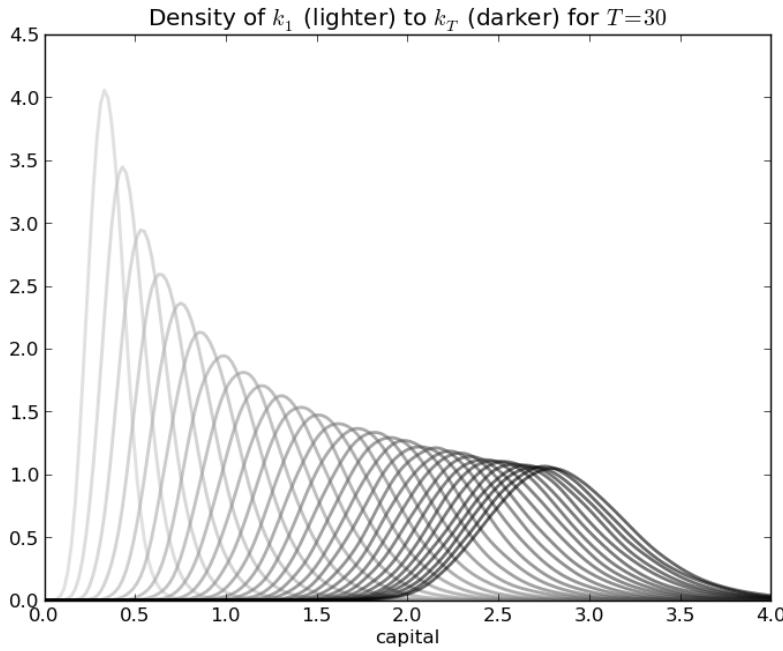
- an object of type LAE that stores the stochastic kernel and the observations
- the value  $y$  as its second argument

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 type — to make vectorization work)

**Example** An example of usage for the stochastic growth model *described above* can be found in `examples/stochasticgrowth.jl`

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

$$X_{t+1} = h(a + \rho X_t + \xi_{t+1}) \quad \text{where} \quad h(x) := x \mathbf{1}\{0 \leq x \leq 1\} + \mathbf{1}\{x > 1\}$$

If you think about it, you will see that for any given  $x \in [0, 1]$ , the conditional distribution of  $X_{t+1}$  given  $X_t = x$  puts positive probability mass on 0 and 1

Hence it cannot be represented as a density

What we can do instead is use cumulative distribution functions (cdfs)

To this end, set

$$G(x, y) := \mathbb{P}\{h(a + \rho x + \xi_{t+1}) \leq y\} \quad (0 \leq x, y \leq 1)$$

This family of cdfs  $G(x, \cdot)$  plays a role analogous to the stochastic kernel in the density case

The distribution dynamics in (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

$$\forall \psi \in \mathcal{D}, \quad \psi P^t \rightarrow \psi^* \quad \text{as } t \rightarrow \infty \quad (3.16)$$

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

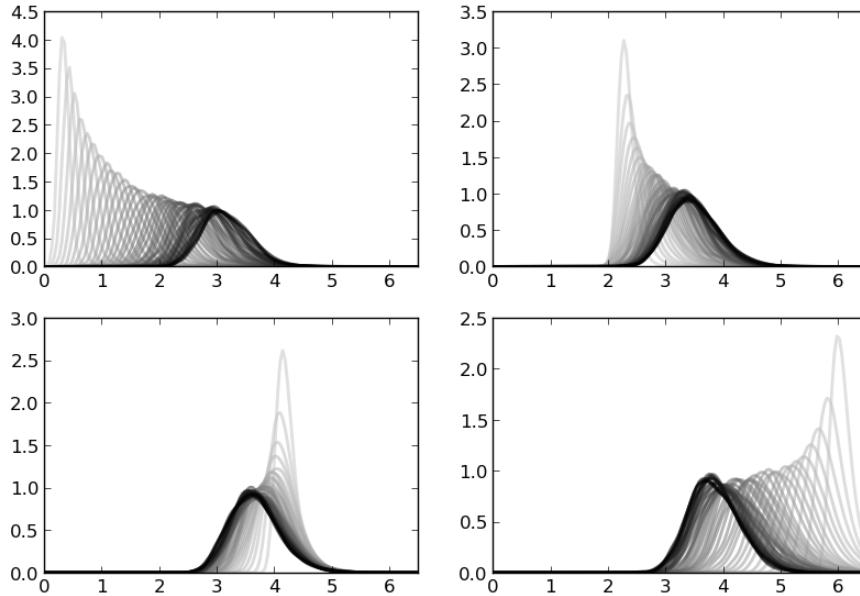
- [\[SLP89\]](#) contains a classic (but slightly outdated) treatment of these topics
- From the mathematical literature, [\[LM94\]](#) and [\[MT09\]](#) give outstanding in depth treatments
- Section 8.1.2 of [EDTC](#) provides detailed intuition, and section 8.3 gives additional references
- [EDTC](#), 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)

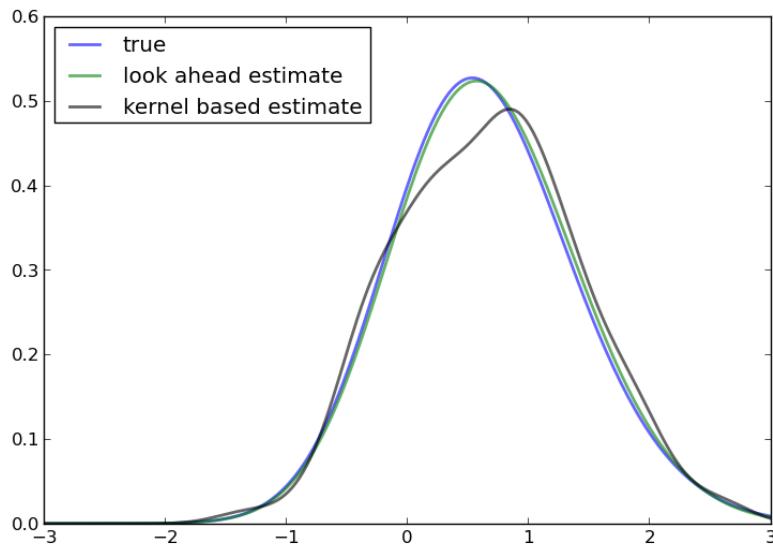
As you can see, the look ahead estimator is a much tighter fit than the kernel density estimator

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 [examples/stochasticgrowth.jl](#)



Use the same parameters

For the four initial distributions, use the beta distribution and shift the random draws as shown below

```
psi_0 = Beta(5.0, 5.0) # Initial distribution
n = 1000
# .... more setup

for i=1:4
    # .... some code
    rand_draws = (rand(psi_0, n) .+ 2.5i) ./ 2
```

**Exercise 3** A common way to compare distributions visually is with boxplots

To illustrate, let's generate three artificial data sets and compare them with a boxplot

```
using PyPlot

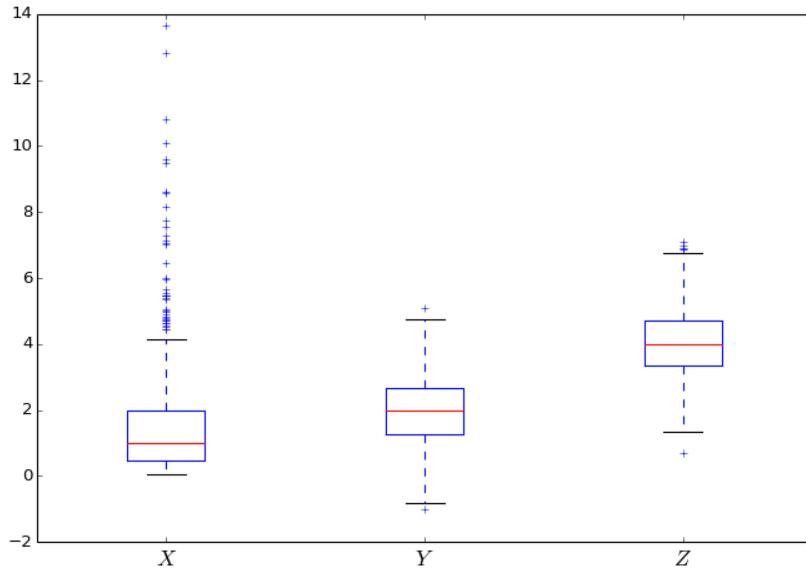
n = 500
n = 500
x = randn(n) # N(0, 1)
x = exp(x) # Map x to lognormal
y = randn(n) + 2.0 # N(2, 1)
z = randn(n) + 4.0 # N(4, 1)

fig, ax = subplots()
ax[:boxplot]([x y z])
ax[:set_xticks]((1, 2, 3))
ax[:set_xlim](-2, 14)
ax[:set_xticklabels]((L"$X$", L"$Y$", L"$Z$"), 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 = 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)

## 3.2 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 nonstororable, so holding assets is the only way to transfer wealth into the future

For the purposes of intuition, it's common to think of the productive unit as a “tree” that produces fruit

Based on this idea, a “Lucas tree” is a claim on the consumption endowment

**Consumers** A representative consumer ranks consumption streams  $\{c_t\}$  according to the time separable utility functional

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{3.21}$$

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.jl` from the `QuantEcon` package

We repeat it here for convenience

```
#=
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
```

```
log y' = alpha log y + sigma epsilon
```

where  $y'$  is a next period  $y$  and  $\epsilon$  is an iid standard normal shock. Hence

```
y' = y^alpha * xi where xi = e^(sigma * epsilon)
```

The distribution  $\phi$  of  $\xi$  is

```
phi = LN(0, sigma^2) where LN means lognormal
```

```
@author : Spencer Lyon <spencer.lyon@nyu.edu>
```

```
@date : 2014-07-05
```

References

-----

```
http://quant-econ.net/jl/markov_asset.html
```

TODO: refactor. Python is much cleaner.

```
=#
```

"""

The Lucas asset pricing model

##### Fields

- `gam::Real` : coefficient of risk aversion in the CRRA utility function

```

- `bet::Real` : Discount factor in (0, 1)
- `alpha::Real` : Correlation coefficient in the shock process
- `sigma::Real` : Volatility of shock process
- `phi::Distribution` : Distribution for shock process
- `grid::AbstractVector` : Grid of points on which to evaluate the prices. Each
point should be non-negative
- `grid_min::Real` : Lower bound on grid
- `grid_max::Real` : Upper bound on grid
- `grid_size::Int` : Number of points in the grid
- `quad_nodes::Vector` : Quadrature nodes for integrating over the shock
- `quad_weights::Vector` : Quadrature weights for integrating over the shock
- `h::Vector` : Storage array for the `h` vector in the lucas operator
"""

type LucasTree
    gam::Real
    bet::Real
    alpha::Real
    sigma::Real
    phi::Distribution
    grid::AbstractVector
    grid_min::Real
    grid_max::Real
    grid_size::Int
    quad_nodes::Vector
    quad_weights::Vector
    h::Vector

    # this needs to be an internal constructor because we need to incompletely
    # initialize the object before we can compute h.
"""

Constructor for LucasTree

##### Arguments

- `gam::Real` : coefficient of risk aversion in the CRRA utility function
- `bet::Real` : Discount factor in (0, 1)
- `alpha::Real` : Correlation coefficient in the shock process
- `sigma::Real` : Volatility of shock process

##### Notes

All other fields of the type are instantiated within the constructor
"""

function LucasTree(gam::Real, bet::Real, alpha::Real, sigma::Real)
    phi = LogNormal(0.0, sigma)
    grid = make_grid(alpha, sigma)
    grid_min, grid_max, grid_size = minimum(grid), maximum(grid), length(grid)
    _int_min, _int_max = exp(-4 * sigma), exp(4 * sigma)
    n, w = qnwlege(21, _int_min, _int_max)

    # create lt object without h
    lt = new(gam, bet, alpha, sigma, phi, grid, grid_min, grid_max, grid_size,
             n, w)

```

```

# initialize h
h = Array(Float64, grid_size)

for (i, y) in enumerate(grid)
    integrand(z) = (y^alpha * z).^(1 - gam)
    h[i] = bet .* integrate(lt, integrand)
end

# now add h to it
lt.h = h
lt
end
end

function make_grid(alpha, sigma)
    grid_size = 100
    if abs(alpha) >= 1
        grid_min, grid_max = 0.0, 10.
    else
        # Set the grid interval to contain most of the mass of the
        # stationary distribution of the consumption endowment
        ssd = sigma / sqrt(1 - alpha^2)
        grid_min, grid_max = exp(-4 * ssd), exp(4 * ssd)
    end
    return linspace_range(grid_min, grid_max, grid_size)
end

function integrate(lt::LucasTree, g::Function, qn::Array=lt.quad_nodes,
                  qw::Array=lt.quad_weights)
    int_func(x) = g(x) .* pdf(lt.phi, x)
    return do_quad(int_func, qn, qw)
end

"""
The approximate Lucas operator, which computes and returns the updated function
Tf on the grid points.
"""

##### Arguments

- `lt::LucasTree` : An instance of the `LucasTree` type
- `f::Vector{Float64}` : A candidate function on  $\mathbb{R}_+$  represented as points on a
grid. It should be the same size as `lt.grid`

##### Returns

- `Tf::Vector{Float64}` : The updated function Tf

"""

function lucas_operator(lt::LucasTree, f::AbstractVector)
    grid, h, alpha, bet = lt.grid, lt.h, lt.alpha, lt.bet

```

```

Tf = similar(f)
Af = CoordInterpGrid(grid, f, BCnearest, InterpLinear)

for (i, y) in enumerate(grid)
    to_integrate(z) = Af[y^alpha * z]
    Tf[i] = h[i] + bet * integrate(lt, to_integrate)
end
return Tf
end

"""
Compute the equilibrium price function associated with Lucas tree `lt`

##### Arguments

- `lt::LucasTree` : An instance of the `LucasTree` type
- `;kwargs...` : other arguments to be passed to `compute_fixed_point`

##### Returns

- `price::Vector{Float64}` : The price at each point in `lt.grid`

"""

function compute_lt_price(lt::LucasTree; kwargs...)
    # Simplify names, set up distribution phi
    grid, grid_size, gam = lt.grid, lt.grid_size, lt.gam

    f_init = zeros(grid) # Initial condition
    func(x) = lucas_operator(lt, x)
    f = compute_fixed_point(func, f_init; kwargs...)

    #  $p(y) = f(y) / u'(y) = f(y) * y^\gamma$ 
    price = f .* grid.^gam

    return price
end

```

An example of usage is given in the docstring and repeated here

```

tree = LucasTree(2, 0.95, 0.90, 0.1)
grid, price_vals = compute_lt_price(tree)

```

Here's the resulting price function

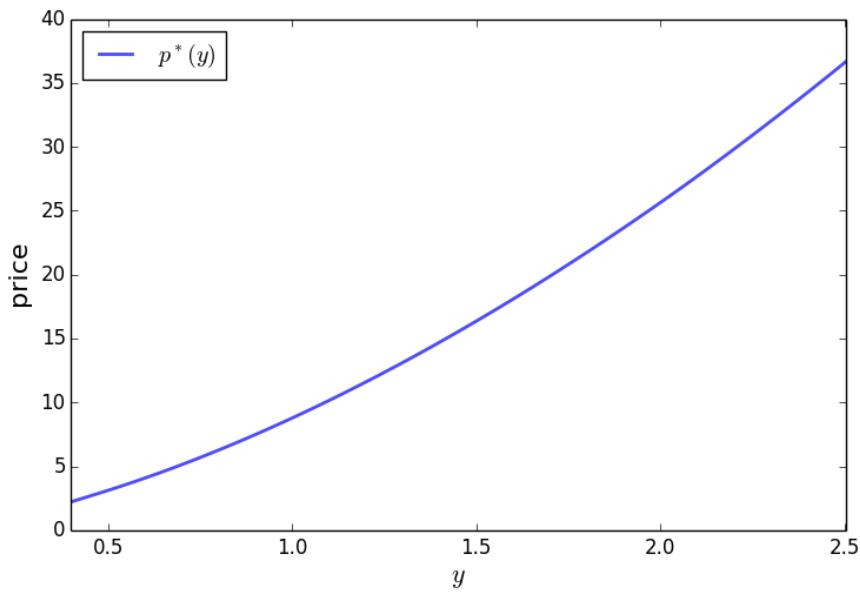
The price is increasing, even if we remove all serial correlation from the endowment process

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

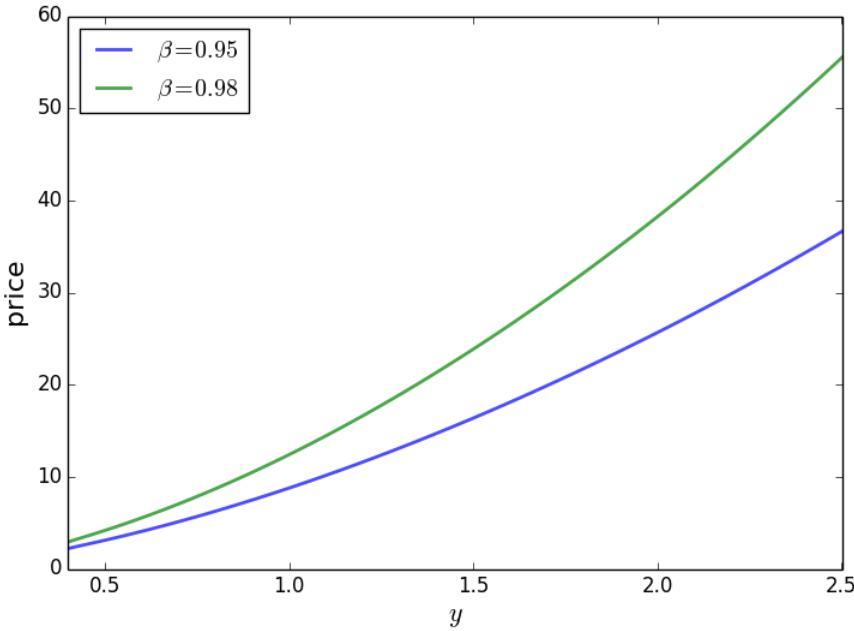
### Solutions

Solution notebook

## 3.3 Modeling Career Choice

### Contents

- *Modeling Career Choice*
  - *Overview*
  - *Model*
  - *Implementation: career.jl*
  - *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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t w_t \quad (3.30)$$

subject to the choice restrictions specified above

Let  $V(\theta, \epsilon)$  denote the value function, which is the maximum of (3.30) over all feasible (career, job) policies, given the initial state  $(\theta, \epsilon)$

The value function obeys

$$V(\theta, \epsilon) = \max\{I, II, III\},$$

where

$$\begin{aligned} I &= \theta + \epsilon + \beta V(\theta, \epsilon) \\ II &= \theta + \int \epsilon' G(d\epsilon') + \beta \int V(\theta, \epsilon') G(d\epsilon') \\ III &= \int \theta' F(d\theta') + \int \epsilon' G(d\epsilon') + \beta \int \int V(\theta', \epsilon') G(d\epsilon') F(d\theta') \end{aligned} \quad (3.31)$$

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 `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 `linspace(0, B, N)`

A very useful family of discrete distributions is the Beta-binomial family, with probability mass function

$$p(k | n, a, b) = \binom{n}{k} \frac{B(k+a, n-k+b)}{B(a, b)}, \quad k = 0, \dots, n$$

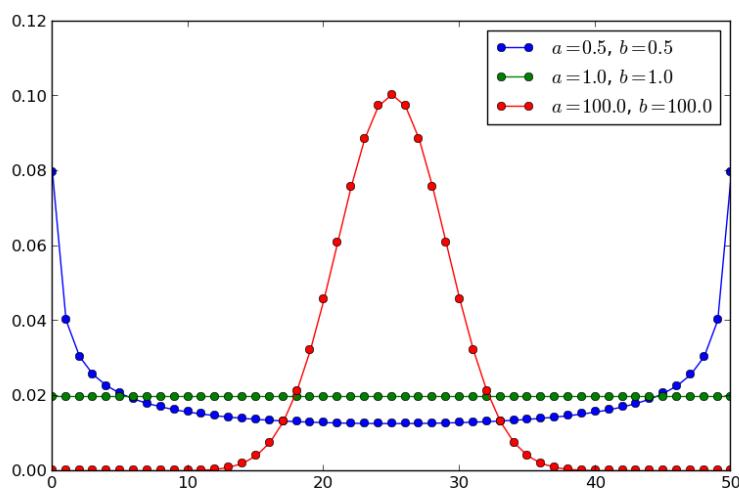
Interpretation:

- 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.jl`

The QuantEcon package provides some code for solving the DP problem described above

See in particular [this file](#), which is repeated here for convenience

```
#=
A type to solve the career / job choice model due to Derek Neal.

@author : Spencer Lyon <spencer.lyon@nyu.edu>

@date: 2014-08-05

References
-----
http://quant-econ.net/jl/career.html
```

```
[Neal1999] Neal, D. (1999). The Complexity of Job Mobility among Young Men, Journal of Labor Economics, 17(2), 237-261.

=#
"""

Career/job choice model fo Derek Neal (1999)

##### Fields

- `beta::Real` : Discount factor in (0, 1)
- `N::Int` : Number of possible realizations of both epsilon and theta
- `B::Real` : upper bound for both epsilon and theta
- `theta::AbstractVector` : A grid of values on [0, B]
- `epsilon::AbstractVector` : A grid of values on [0, B]
- `F_probs::AbstractVector` : The pdf of each value associated with of F
- `G_probs::AbstractVector` : The pdf of each value associated with of G
- `F_mean::Real` : The mean of the distribution F
- `G_mean::Real` : The mean of the distribution G

"""

type CareerWorkerProblem
    beta::Real
    N::Int
    B::Real
    theta::AbstractVector
    epsilon::AbstractVector
    F_probs::AbstractVector
    G_probs::AbstractVector
    F_mean::Real
    G_mean::Real
end

"""

Constructor with default values for `CareerWorkerProblem`

##### Arguments

- `beta::Real(0.95)` : Discount factor in (0, 1)
- `B::Real(5.0)` : upper bound for both epsilon and theta
- `N::Real(50)` : Number of possible realizations of both epsilon and theta
- `F_a::Real(1), F_b::Real(1)` : Parameters of the distribution F
- `G_a::Real(1), G_b::Real(1)` : Parameters of the distribution F

##### Notes

$(_-_kward_note)
"""

function CareerWorkerProblem(beta::Real=0.95, B::Real=5.0, N::Real=50,
                           F_a::Real=1, F_b::Real=1, G_a::Real=1,
                           G_b::Real=1)
    theta = linspace(0, B, N)
    epsilon = copy(theta)
    F_probs::Vector{Float64} = pdf(BetaBinomial(N-1, F_a, F_b))
end
```

```

G_probs::Vector{Float64} = pdf(BetaBinomial(N_1, G_a, G_b))
F_mean = sum(theta .* F_probs)
G_mean = sum(epsilon .* G_probs)
CareerWorkerProblem(beta, N, B, theta, epsilon, F_probs, G_probs,
                     F_mean, G_mean)
end

# create kwarg version
function CareerWorkerProblem(;beta::Real=0.95, B::Real=5.0, N::Real=50,
                             F_a::Real=1, F_b::Real=1, G_a::Real=1,
                             G_b::Real=1)
    CareerWorkerProblem(beta, B, N, F_a, F_b, G_a, G_b)
end

"""
$(_-_bellman_main_docstring).

##### Arguments

- `cp::CareerWorkerProblem` : Instance of `CareerWorkerProblem`
- `v::Matrix` : Current guess for the value function
- `out::Matrix` : Storage for output
- `;ret_policy::Bool(false)` : Toggles return of value or policy functions

##### Returns

None, `out` is updated in place. If `ret_policy == true` `out` is filled with the
policy function, otherwise the value function is stored in `out`.

"""
function bellman_operator!(cp::CareerWorkerProblem, v::Array, out::Array;
                           ret_policy=false)
    # new life. This is a function of the distribution parameters and is
    # always constant. No need to recompute it in the loop
    v3 = (cp.G_mean + cp.F_mean + cp.beta .*
          cp.F_probs' * v * cp.G_probs)[1] # don't need 1 element array

    for j=1:cp.N
        for i=1:cp.N
            # stay put
            v1 = cp.theta[i] + cp.epsilon[j] + cp.beta * v[i, j]

            # new job
            v2 = (cp.theta[i] .+ cp.G_mean .+ cp.beta .*
                  v[i, :] * cp.G_probs)[1] # don't need a single element array

            if ret_policy
                if v1 > max(v2, v3)
                    action = 1
                elseif v2 > max(v1, v3)
                    action = 2
                else
                    action = 3
                end
            else
                out[i, j] = v2
            end
        end
    end
end

```

```

        end
        out[i, j] = action
    else
        out[i, j] = max(v1, v2, v3)
    end
end
end

function bellman_operator(cp::CareerWorkerProblem, v::Array; ret_policy=false)
    out = similar(v)
    bellman_operator!(cp, v, out, ret_policy=ret_policy)
    return out
end

"""
$(_-_greedy_main_docstring).

##### Arguments

- `cp::CareerWorkerProblem` : Instance of `CareerWorkerProblem`
- `v::Matrix` : Current guess for the value function
- `out::Matrix` : Storage for output

##### Returns

None, `out` is updated in place to hold the policy function

"""
function get_greedy!(cp::CareerWorkerProblem, v::Array, out::Array)
    bellman_operator!(cp, v, out, ret_policy=true)
end

function get_greedy(cp::CareerWorkerProblem, v::Array)
    bellman_operator(cp, v, ret_policy=true)
end

```

The code defines

- a type `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 (3.31), replacing  $V$  with  $v$

The default probability distributions in `CareerWorkerProblem` correspond to discrete uniform distributions (see *the Beta-binomial figure*)

In fact all our default settings correspond to the version studied in [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

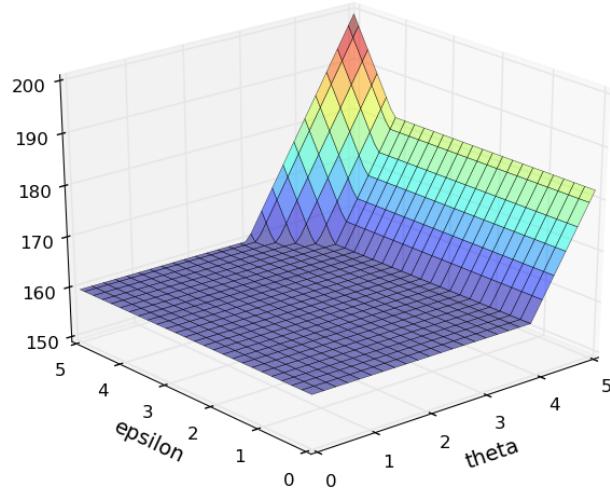


Fig. 3.1: Value function with uniform probabilities

The code used to produce this plot was [examples/career\\_vf\\_plot.jl](#)

The optimal policy can be represented as follows (see *Exercise 3* for code)

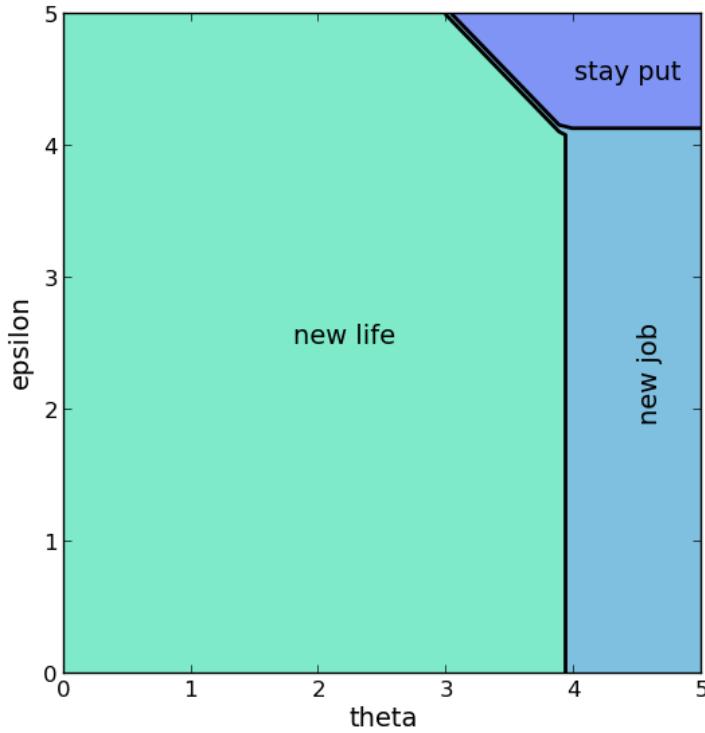
Interpretation:

- 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 type `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 type `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

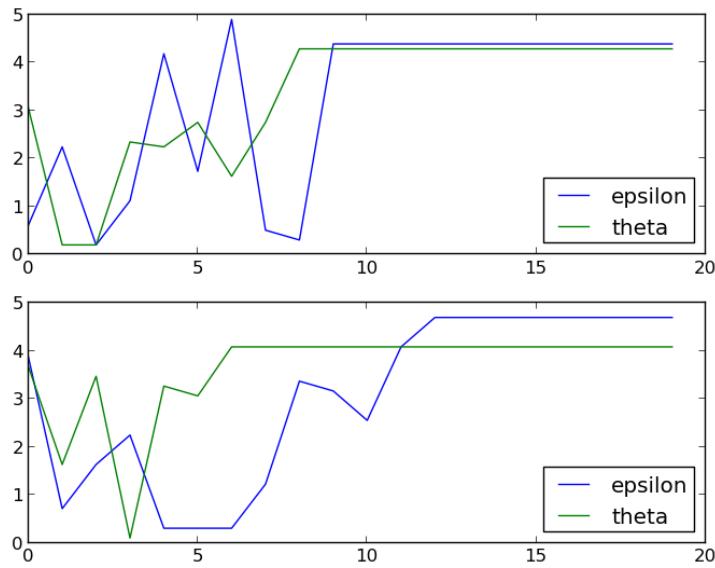
Evidently, the worker's job becomes permanent if and only if  $(\theta_t, \epsilon_t)$  enters the "stay put" region of  $(\theta, \epsilon)$  space

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 `PyPlot.jl` to produce the different shadings.

Now set `G_a = G_b = 100` and generate a new figure with these parameters. Interpret.

### Solutions

[Solution notebook](#)

## 3.4 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

$$x_{t+1} = (1 - b_{t+1})G(x_t, \phi_t) + b_{t+1} \max\{G(x_t, \phi_t), U_{t+1}\} \quad (3.32)$$

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 (3.32), the Bellman equation for this problem can be written as

$$V(x) = \max_{s+\phi \leq 1} \left\{ x(1 - s - \phi) + \beta(1 - \pi(s))V[G(x, \phi)] + \beta\pi(s) \int V[G(x, \phi) \vee u]F(du) \right\}. \quad (3.33)$$

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 (3.32) 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.jl`, which is repeated here for convenience

```
#=
@autho r : Spencer Lyon <spencer.lyon@nyu.edu>
@date: 2014-06-27
References
-----
Simple port of the file quantecon.models.jv

http://quant-econ.net/jl/jv.html
=#
# TODO: the three lines below will allow us to use the non brute-force
#       approach in bellman operator. I have commented it out because
#       I am waiting on a simple constrained optimizer to be written in
#       pure Julia

# using PyCall
# @pyimport scipy.optimize as opt
# minimize = opt.minimize

epsilon = 1e-4 # a small number, used in optimization routine

"""
A Jovanovic-type model of employment with on-the-job search.

The value function is given by

\\[V(x) = \max_{\phi, s} w(x, \phi, s)\\]

for

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)]

where

* `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` : Random variable with distribution F -- new draw of human capita

##### Fields
```

```

- `A::Real` : Parameter in human capital transition function
- `alpha::Real` : Parameter in human capital transition function
- `bet::Real` : Discount factor in (0, 1)
- `x_grid::FloatRange` : Grid for potential levels of x
- `G::Function` : Transition `function` for human capital
- `pi_func::Function` : `function` mapping search effort to the probability of
getting a new job offer
- `F::UnivariateDistribution` : A univariate distribution from which the value
of new job offers is drawn
- `quad_nodes::Vector` : Quadrature nodes for integrating over phi
- `quad_weights::Vector` : Quadrature weights for integrating over phi

"""

type JvWorker
    A::Real
    alpha::Real
    bet::Real
    x_grid::FloatRange
    G::Function
    pi_func::Function
    F::UnivariateDistribution
    quad_nodes::Vector
    quad_weights::Vector
end

"""

Constructor with default values for `JvWorker`


##### Arguments

- `A::Real(1.4)` : Parameter in human capital transition function
- `alpha::Real(0.6)` : Parameter in human capital transition function
- `bet::Real(0.96)` : Discount factor in (0, 1)
- `grid_size::Int(50)` : Number of points in discrete grid for `x`


##### Notes

$(_-_kward_note)

"""

function JvWorker(A=1.4, alpha=0.6, bet=0.96, grid_size=50)
    G(x, phi) = A .* (x .* phi).^alpha
    pi_func = sqrt
    F = Beta(2, 2)

    # integration bounds
    a, b, = quantile(F, 0.005), quantile(F, 0.995)

    # quadrature nodes/weights
    nodes, weights = qnwlege(21, a, b)

    # 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^(1.0 / (1.0 - alpha)), quantile(F, 1 - epsilon))

# range for linspace(epsilon, grid_max, grid_size). Needed for
# CoordInterpGrid below
x_grid = linspace_range(epsilon, grid_max, grid_size)

JvWorker(A, alpha, bet, x_grid, G, pi_func, F, nodes, weights)
end

# make kwarg version
JvWorker(;A=1.4, alpha=0.6, bet=0.96, grid_size=50) = JvWorker(A, alpha, bet,
grid_size)

# TODO: as of 2014-08-14 there is no simple constrained optimizer in Julia
#       so, we default to the brute force gridsearch approach for this
#       problem

# NOTE: this function is not type stable because it returns either
#       Array{Float64, 2} or (Array{Float64, 2}, Array{Float64, 2})
#       depending on the value of ret_policies. This is probably not a
#       huge deal, but it is something to be aware of
"""
$(____bellman_main_docstring).

##### Arguments

- `jv::JvWorker` : Instance of `JvWorker`
- `V::Vector` : Current guess for the value function
- `out::Union(Vector, Tuple{Vector, Vector})` : Storage for output. Note that
there are two policy rules, but one value function
- `;brute_force::Bool(true)` : Whether to use a brute force grid search
algorithm or a solver from scipy.
- `;ret_policy::Bool(false)` : Toggles return of value or policy functions

##### Returns

None, `out` is updated in place. If `ret_policy == true` `out` is filled with the
policy function, otherwise the value function is stored in `out`.

##### Notes

Currently, the `brute_force` parameter must be `true`. We are waiting for a
constrained optimization routine to emerge in pure Julia. Once that happens,
we will re-activate this option.

"""
function bellman_operator!(jv::JvWorker, V::Vector,
                           out::Union(Vector, @compat Tuple{Vector, Vector});
                           brute_force=true, ret_policies=false)

    if !(brute_force)

```

```

m = "Only brute_force method active now. Waiting on a pure julia"
m *= " constrained optimization routine to disable"
error(m)
end
# simplify notation
G, pi_func, F, bet = jv.G, jv.pi_func, jv.F, jv.bet
nodes, weights = jv.quad_nodes, jv.quad_weights

# prepare interpoland of value function
Vf = CoordInterpGrid(jv.x_grid, V, BCnearest, InterpLinear)

# instantiate variables so they are available outside loop and exist
# within it
if ret_policies
    if !(typeof(out) <: @compat Tuple{Vector, Vector})
        msg = "You asked for policies, but only provided one output array"
        msg *= "\nthere are two policies so two arrays must be given"
        error(msg)
    end
    s_policy, phi_policy = out[1], out[2]
else
    c1(z) = 1.0 - sum(z)
    c2(z) = z[1] - epsilon
    c3(z) = z[2] - epsilon
    guess = (0.2, 0.2)
    constraints = [@compat Dict("type" => "ineq", "fun"=> i) for i in [c1, c2, c3]]
    if typeof(out) <: Tuple
        msg = "Multiple output arrays given. There is only one value"
        msg = " function.\nDid you mean to pass ret_policies=true?"
        error(msg)
    end
    new_V = out
end

# instantiate the linesearch variables if we need to
if brute_force
    max_val = -1.0
    cur_val = 0.0
    max_s = 1.0
    max_phi = 1.0
    search_grid = linspace(epsilon, 1.0, 15)
end

for (i, x) in enumerate(jv.x_grid)

    function w(z)
        s, phi = z
        h(u) = Vf[max(G(x, phi), u)] .* pdf(F, u)
        integral = do_quad(h, nodes, weights)
        q = pi_func(s) * integral + (1.0 - pi_func(s)) * Vf[G(x, phi)]

        return - x * (1.0 - phi - s) - bet * q
    end

```

```

    end

    if brute_force
        for s in search_grid
            for phi in search_grid
                if s + phi <= 1.0
                    cur_val = -w((s, phi))
                else
                    cur_val = -1.0
                end
                if cur_val > max_val
                    max_val, max_s, max_phi = cur_val, s, phi
                end
            end
        end
    else
        max_s, max_phi = minimize(w, guess, constraints=constraints;
                                    disp=0, method="SLSQP")["x"]

        max_val = -w((max_s, max_phi), x, a, b, Vf, jv)

    end

    if ret_policies
        s_policy[i], phi_policy[i] = max_s, max_phi
    else
        new_V[i] = max_val
    end
end
end

function bellman_operator(jv::JvWorker, V::Vector; brute_force=true,
                         ret_policies=false)
    if ret_policies
        out = (similar(V), similar(V))
    else
        out = similar(V)
    end
    bellman_operator!(jv, V, out, brute_force=brute_force,
                      ret_policies=ret_policies)
    return out
end

"""
$(_-_greedy_main_docstring).

##### Arguments

- `cp::CareerWorkerProblem` : Instance of `CareerWorkerProblem`
- `v::Vector` : Current guess for the value function
- `out::Tuple(Vector, Vector)` : Storage for output of policy rule

```

```
##### Returns

None, `out` is updated in place to hold the policy function

"""

function get_greedy!(jv::JvWorker, V::Vector, out::Compat.Tuple{Vector, Vector};
    brute_force=true)
    bellman_operator!(jv, V, out, ret_policies=true)
end

function get_greedy(jv::JvWorker, V::Vector; brute_force=true)
    bellman_operator(jv, V, ret_policies=true)
end
```

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 type 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

$$w(s, \phi) := - \left\{ x(1 - s - \phi) + \beta(1 - \pi(s))V[G(x, \phi)] + \beta\pi(s) \int V[G(x, \phi) \vee u]F(du) \right\} \quad (3.34)$$

Here we are minimizing instead of maximizing to fit with SciPy's optimization routines

When we represent  $V$ , it will be with a Julia array `V` giving values on grid `x_grid`

But to evaluate the right-hand side of (3.34), 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 (3.34).

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 `examples/jv_test.jl` from the [main repository](#) and looks as follows

```
s_policy, phi_policy = bellman_operator(wp, V, return_policies=true)

# === plot policies === #
fig, ax = subplots()
ax[:set_xlim](0, maximum(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

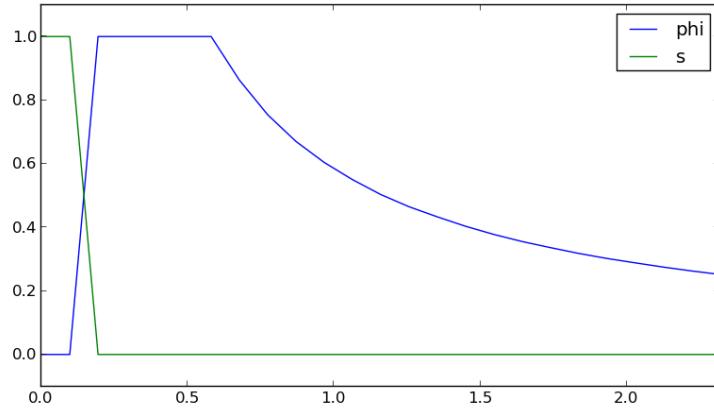


Fig. 3.2: Optimal policies

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 (3.32) 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 = linspace(0, plot_grid_max, plot_grid_size)
fig, ax = 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](#)

# 3.5 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

$$V(w) = \max \left\{ \frac{w}{1-\beta}, c + \beta \int V(w') h(w') dw' \right\} \quad (3.35)$$

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

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t)g(w_{t+1})} \quad (3.36)$$

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 (3.36) 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

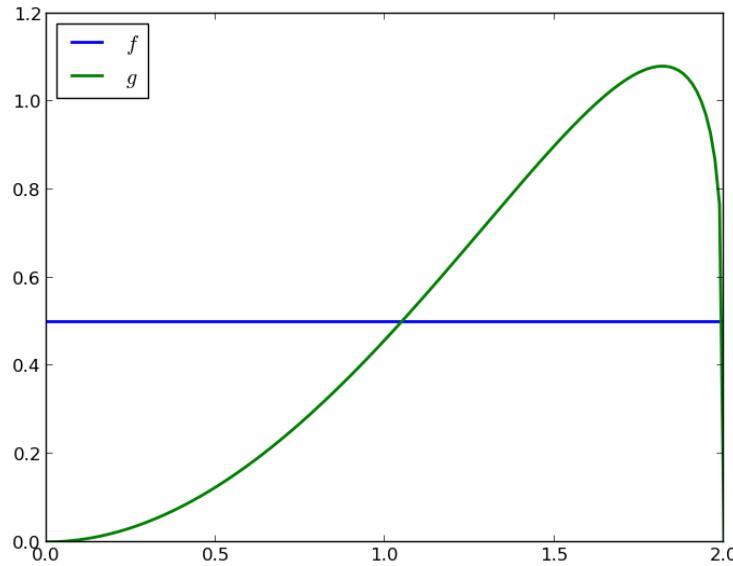
$$V(w, \pi) = \max \left\{ \frac{w}{1 - \beta}, c + \beta \int V(w', \pi') h_\pi(w') dw' \right\} \quad \text{where} \quad \pi' = q(w', \pi) \quad (3.37)$$

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

- $f = \text{Beta}(1, 1)$  and  $g = \text{Beta}(3, 1.2)$
- $\beta = 0.95$  and  $c = 0.6$

The densities  $f$  and  $g$  have the following shape



**Looking Forward** What kind of optimal policy might result from (3.37) 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.jl` contained in the `QuantEcon` package

The code is as follows

```
#=
Solves the "Offer Distribution Unknown" Model by value function
iteration and a second faster method discussed in the corresponding
quantecon lecture.

@author : Spencer Lyon <spencer.lyon@nyu.edu>

@date: 2014-08-14

References
-----
http://quant-econ.net/jl/odu.html

=#
"""

Unemployment/search problem where offer distribution is unknown

##### Fields

- `bet::Real` : Discount factor on (0, 1)
- `c::Real` : Unemployment compensation
```

```

- `F::Distribution` : Offer distribution `F`
- `G::Distribution` : Offer distribution `G`
- `f::Function` : The pdf of `F`
- `g::Function` : The pdf of `G`
- `n_w::Int` : Number of points on the grid for w
- `w_max::Real` : Maximum wage offer
- `w_grid::AbstractVector` : Grid of wage offers w
- `n_pi::Int` : Number of points on grid for pi
- `pi_min::Real` : Minimum of pi grid
- `pi_max::Real` : Maximum of pi grid
- `pi_grid::AbstractVector` : Grid of probabilities pi
- `quad_nodes::Vector` : Notes for quadrature ofer offers
- `quad_weights::Vector` : Weights for quadrature ofer offers

"""

type SearchProblem
    bet::Real
    c::Real
    F::Distribution
    G::Distribution
    f::Function
    g::Function
    n_w::Int
    w_max::Real
    w_grid::AbstractVector
    n_pi::Int
    pi_min::Real
    pi_max::Real
    pi_grid::AbstractVector
    quad_nodes::Vector
    quad_weights::Vector
end

"""

Constructor for `SearchProblem` with default values

##### Arguments

- `bet::Real(0.95)` : Discount factor in (0, 1)
- `c::Real(0.6)` : Unemployment compensation
- `F_a::Real(1), F_b::Real(1)` : Parameters of `Beta` distribution for `F`
- `G_a::Real(3), G_b::Real(1.2)` : Parameters of `Beta` distribution for `G`
- `w_max::Real(2)` : Maximum of wage offer grid
- `w_grid_size::Int(40)` : Number of points in wage offer grid
- `pi_grid_size::Int(40)` : Number of points in probability grid

##### Notes

$(_-_kward_note)

"""

function SearchProblem(bet=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)

```

```

F = Beta(F_a, F_b)
G = Beta(G_a, G_b)

# NOTE: the x./w_max)./w_max in these functions makes our dist match
#      the scipy one with scale=w_max given
f(x) = pdf(F, x./w_max)./w_max
g(x) = pdf(G, x./w_max)./w_max

pi_min = 1e-3 # avoids instability
pi_max = 1 - pi_min

w_grid = linspace_range(0, w_max, w_grid_size)
pi_grid = linspace_range(pi_min, pi_max, pi_grid_size)

nodes, weights = qnwlege(21, 0.0, w_max)

SearchProblem(bet, c, F, G, f, g,
              w_grid_size, w_max, w_grid,
              pi_grid_size, pi_min, pi_max, pi_grid, nodes, weights)
end

# make kwarg version
function SearchProblem(;bet=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)
    SearchProblem(bet, c, F_a, F_b, G_a, G_b, w_max, w_grid_size,
                  pi_grid_size)
end

function q(sp::SearchProblem, w, pi_val)
    new_pi = 1.0 ./ (1 + ((1 - pi_val) .* sp.g(w)) ./ (pi_val .* sp.f(w)))

    # Return new_pi when in [pi_min, pi_max] and else end points
    return clamp(new_pi, sp.pi_min, sp.pi_max)
end

"""
$(_-_bellman_main_docstring).

##### Arguments

- `sp::SearchProblem` : Instance of `SearchProblem`
- `v::Matrix` : Current guess for the value function
- `out::Matrix` : Storage for output.
- `;ret_policy::Bool(false)` : Toggles return of value or policy functions

##### Returns

None, `out` is updated in place. If `ret_policy == true` `out` is filled with the
policy function, otherwise the value function is stored in `out`.

"""
function bellman_operator!(sp::SearchProblem, v::Matrix, out::Matrix;
                           ret_policy::Bool=false)

```

```

# Simplify names
f, g, bet, c = sp.f, sp.g, sp.bet, sp.c
nodes, weights = sp.quad_nodes, sp.quad_weights

vf = CoordInterpGrid((sp.w_grid, sp.pi_grid), v, BCnan, InterpLinear)

# set up quadrature nodes/weights
# q_nodes, q_weights = qnwlege(21, 0.0, sp.w_max)

for w_i=1:sp.n_w
    w = sp.w_grid[w_i]

    # calculate v1
    v1 = w / (1 - bet)

    for pi_j=1:sp.n_pi
        _pi = sp.pi_grid[pi_j]

        # calculate v2
        function integrand(m)
            quad_out = similar(m)
            for i=1:length(m)
                mm = m[i]
                quad_out[i] = vf[mm, q(sp, mm, _pi)] * (_pi*f(mm) +
                                                (1-_pi)*g(mm))
            end
            return quad_out
        end
        integral = do_quad(integrand, nodes, weights)
        # integral = do_quad(integrand, q_nodes, q_weights)
        v2 = c + bet * integral

        # return policy if asked for, otherwise return max of values
        out[w_i, pi_j] = ret_policy ? v1 > v2 : max(v1, v2)
    end
end
return out
end

function bellman_operator(sp::SearchProblem, v::Matrix;
                           ret_policy::Bool=false)
    out_type = ret_policy ? Bool : Float64
    out = Array(out_type, sp.n_w, sp.n_pi)
    bellman_operator!(sp, v, out, ret_policy=ret_policy)
end

"""
$(_-_greedy_main_docstring).

##### Arguments

- `sp::SearchProblem` : Instance of `SearchProblem`

```

```

- `v::Matrix`: Current guess for the value function
- `out::Matrix` : Storage for output

##### Returns

None, `out` is updated in place to hold the policy function

"""

function get_greedy!(sp::SearchProblem, v::Matrix, out::Matrix)
    bellman_operator!(sp, v, out, ret_policy=true)
end

get_greedy(sp::SearchProblem, v::Matrix) = bellman_operator(sp, v,
                                                               ret_policy=true)

"""

Updates the reservation wage function guess phi via the operator Q.

##### Arguments

- `sp::SearchProblem` : Instance of `SearchProblem`
- `phi::Vector` : Current guess for phi
- `out::Vector` : Storage for output

##### Returns

None, `out` is updated in place to hold the updated levels of phi
"""

function res_wage_operator!(sp::SearchProblem, phi::Vector, out::Vector)
    # Simplify name
    f, g, bet, c = sp.f, sp.g, sp.bet, sp.c

    # Construct interpolator over pi_grid, given phi
    phi_f = CoordInterpGrid(sp.pi_grid, phi, BCnearest, InterpLinear)

    # set up quadrature nodes/weights
    q_nodes, q_weights = qnwlege(7, 0.0, sp.w_max)

    for (i, _pi) in enumerate(sp.pi_grid)
        integrand(x) = max(x, phi_f[q(sp, x, _pi)]).*(_pi*f(x) + (1-_pi)*g(x))
        integral = do_quad(integrand, q_nodes, q_weights)
        out[i] = (1 - bet)*c + bet*integral
    end
end

"""

Updates the reservation wage function guess phi via the operator Q.

See the documentation for the mutating method of this function for more details
on arguments
"""

function res_wage_operator(sp::SearchProblem, phi::Vector)

```

```

out = similar(phi)
res_wage_operator!(sp, phi, out)
return out
end

```

The type `SearchProblem` is used to store parameters and methods needed to compute optimal actions

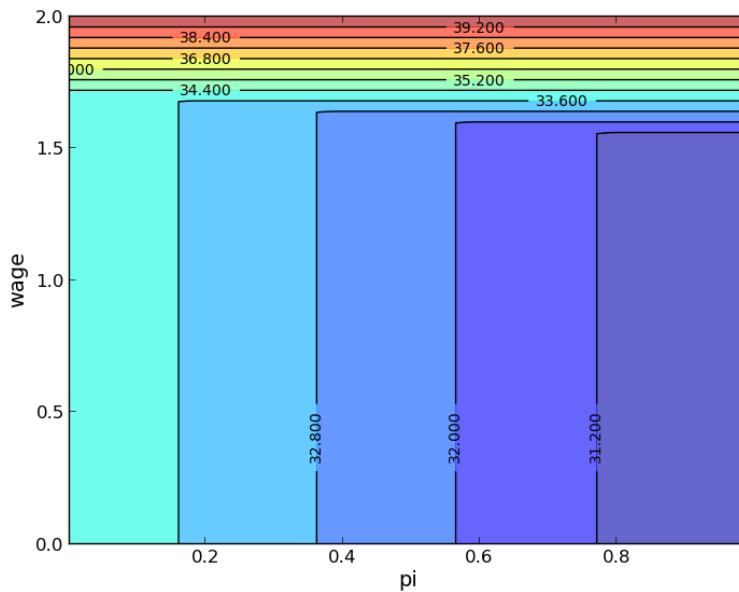
The Bellman operator is implemented as the method `bellman_operator()`, while `get_greedy()` computes an approximate optimal policy from a guess  $v$  of the value function

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:



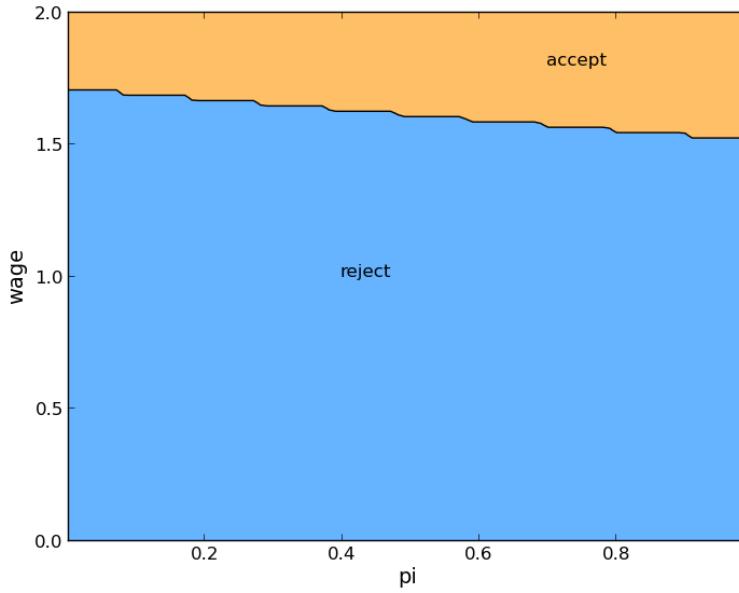
The optimal policy:

Code for producing these figures can be found in file `examples/odu_vfi_plots.jl` from the [main 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 (3.37) have equal value:

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int V(w', \pi') h_\pi(w') dw' \quad (3.38)$$

Together, (3.37) and (3.38) give

$$V(w, \pi) = \max \left\{ \frac{w}{1-\beta}, \frac{\bar{w}(\pi)}{1-\beta} \right\} \quad (3.39)$$

Combining (3.38) and (3.39), 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

$$\bar{w}(\pi) = (1 - \beta)c + \beta \int \max \{w', \bar{w} \circ q(w', \pi)\} h_\pi(w') dw' \quad (3.40)$$

Equation (3.40) 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

$$(Q\psi)(\pi) = (1 - \beta)c + \beta \int \max \{w', \psi \circ q(w', \pi)\} h_\pi(w') dw' \quad (3.41)$$

Comparing (3.40) and (3.41), 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 (3.40) and vice versa

Moreover, for any  $\psi, \phi \in b[0, 1]$ , basic algebra and the triangle inequality for integrals tells us that

$$|(Q\psi)(\pi) - (Q\phi)(\pi)| \leq \beta \int |\max \{w', \psi \circ q(w', \pi)\} - \max \{w', \phi \circ q(w', \pi)\}| h_\pi(w') dw' \quad (3.42)$$

Working case by case, it is easy to check that for real numbers  $a, b, c$  we always have

$$|\max\{a, b\} - \max\{a, c\}| \leq |b - c| \quad (3.43)$$

Combining (3.42) and (3.43) yields

$$|(Q\psi)(\pi) - (Q\phi)(\pi)| \leq \beta \int |\psi \circ q(w', \pi) - \phi \circ q(w', \pi)| h_\pi(w') dw' \leq \beta \|\psi - \phi\| \quad (3.44)$$

Taking the supremum over  $\pi$  now gives us

$$\|Q\psi - Q\phi\| \leq \beta \|\psi - \phi\| \quad (3.45)$$

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.jl` 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](#)

## 3.6 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

$$c_t + a_{t+1} \leq Ra_t + z_t, \quad c_t \geq 0, \quad a_t \geq -b \quad t = 0, 1, \dots \quad (3.46)$$

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 (3.46), 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

$$V(a, z) := \sup \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\} \quad (3.47)$$

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 (3.47)

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

$$u'(c_t) = \max \{ \beta R \mathbb{E}_t [u'(c_{t+1})], u'(Ra_t + z_t + b) \} \quad (3.48)$$

and the transversality condition

$$\lim_{t \rightarrow \infty} \beta^t \mathbb{E} [u'(c_t) a_{t+1}] = 0. \quad (3.49)$$

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 (3.48) and (3.49), 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 (3.48) to make it a statement about functions rather than random variables

In particular, consider the functional equation

$$u' \circ c(a, z) = \max \left\{ \gamma \int u' \circ c \{ Ra + z - c(a, z), \dot{z} \} \Pi(z, d\dot{z}), u'(Ra + z + b) \right\} \quad (3.50)$$

where  $\gamma := \beta R$  and  $u' \circ c(s) := u'(c(s))$

Equation (3.50) 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

$$u'(t) = \max \left\{ \gamma \int u' \circ c \{ Ra + z - t, \dot{z} \} \Pi(z, d\dot{z}), u'(Ra + z + b) \right\} \quad (3.51)$$

where

$$J(a, z) := \{t \in \mathbb{R} : \min Z \leq t \leq Ra + z + b\} \quad (3.52)$$

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 (3.50) 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 (3.51)

### Value function iteration

The Bellman operator for this problem is given by

$$Tv(a, z) = \max_{0 \leq c \leq Ra + z + b} \left\{ u(c) + \beta \int v(Ra + z - c, \dot{z}) \Pi(z, d\dot{z}) \right\} \quad (3.53)$$

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.jl` from QuantEcon provides implementations of both VFI and PFI

The code is repeated here and a description and clarifications are given below

```
#=
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.

@author : Spencer Lyon <spencer.lyon@nyu.edu>

@date: 2014-08-18

References
-----
http://quant-econ.net/jl/ifp.html

=#
# using PyCall
# @pyimport scipy.optimize as opt
# brentq = opt.brentq

"""
Income fluctuation problem

##### Fields

- `u::Function` : Utility `function`
- `du::Function` : Marginal utility `function`
- `r::Real` : Strictly positive interest rate
- `R::Real` : The interest rate plus 1 (strictly greater than 1)
- `bet::Real` : Discount rate in (0, 1)
- `b::Real` : The borrowing constraint
- `Pi::Matrix` : Transition matrix for `z`
- `z_vals::Vector` : Levels of productivity
- `asset_grid::AbstractVector` : Grid of asset values
"""
type ConsumerProblem
    u::Function
    du::Function
```

```

r::Real
R::Real
bet::Real
b::Real
Pi::Matrix
z_vals::Vector
asset_grid::AbstractVector
end

"Marginal utility for log utility function"
default_du{T <: Real}(x::T) = 1.0 / x

"""
Constructor with default values for `ConsumerProblem`


##### Arguments

- `r::Real(0.01)` : Strictly positive interest rate
- `bet::Real(0.96)` : Discount rate in (0, 1)
- `Pi::Matrix{Float64}([0.6 0.4; 0.05 0.95])` : Transition matrix for `z`
- `z_vals::Vector{Float64}([0.5, 1.0])` : Levels of productivity
- `b::Real(0.0)` : Borrowing constraint
- `grid_max::Real(16)` : Maximum in grid for asset holdings
- `grid_size::Int(50)` : Number of points in grid for asset holdings
- `u::Function(log)` : Utility `function`
- `du::Function(x->1/x)` : Marginal utility `function`


##### Notes

$(_-_kward_note)

"""

function ConsumerProblem(r=0.01, bet=0.96, Pi=[0.6 0.4; 0.05 0.95],
                        z_vals=[0.5, 1.0], b=0.0, grid_max=16, grid_size=50,
                        u=log, du=default_du)
    R = 1 + r
    asset_grid = linspace_range(-b, grid_max, grid_size)

    ConsumerProblem(u, du, r, R, bet, b, Pi, z_vals, asset_grid)
end

# make kward version
function ConsumerProblem(;r=0.01, beta=0.96, Pi=[0.6 0.4; 0.05 0.95],
                        z_vals=[0.5, 1.0], b=0.0, grid_max=16, grid_size=50,
                        u=log, du=x -> 1./x)
    ConsumerProblem(r, beta, Pi, z_vals, b, grid_max, grid_size, u, du)
end

"""

$(_-_bellman_main_docstring).

##### Arguments

```

```

- `cp::ConsumerProblem` : Instance of `ConsumerProblem`
- `v::Matrix` : Current guess for the value function
- `out::Matrix` : Storage for output
- `;ret_policy::Bool(false)` : Toggles return of value or policy functions

##### Returns

None, `out` is updated in place. If `ret_policy == true` `out` is filled with the
policy function, otherwise the value function is stored in `out`.

"""
function bellman_operator!(cp::ConsumerProblem, V::Matrix, out::Matrix;
                           ret_policy::Bool=false)
    # simplify names, set up arrays
    R, Pi, bet, u, b = cp.R, cp.Pi, cp.bet, cp.u, cp.b
    asset_grid, z_vals = cp.asset_grid, cp.z_vals

    new_V = similar(V)
    new_c = similar(V)

    z_idx = 1:length(z_vals)

    # Linear interpolation of V along the asset grid
    vf(a, i_z) = CoordInterpGrid(asset_grid, V[:, i_z], BCnearest,
                                  InterpLinear)[a]

    # compute lower_bound for optimization
    opt_lb = minimum(z_vals) - 1e-5

    # solve for RHS of Bellman equation
    for (i_z, z) in enumerate(z_vals)
        for (i_a, a) in enumerate(asset_grid)

            function obj(c)
                y = sum([vf(R*a+z-c, j) * Pi[i_z, j] for j=z_idx])
                return -u(c) - bet * y
            end
            res = optimize(obj, opt_lb, R.*a.+z.+b)
            c_star = res.minimum
            if ret_policy
                out[i_a, i_z] = c_star
            else
                out[i_a, i_z] = - obj(c_star)
            end
        end
    end
end

function bellman_operator(cp::ConsumerProblem, V::Matrix; ret_policy=false)
    out = similar(V)
    bellman_operator!(cp, V, out, ret_policy=ret_policy)
    return out
end

```

```

"""
$(_-_greedy_main_docstring).

##### Arguments

- `cp::CareerWorkerProblem` : Instance of `CareerWorkerProblem`
- `v::Matrix` : Current guess for the value function
- `out::Matrix` : Storage for output

##### Returns

None, `out` is updated in place to hold the policy function

"""

function get_greedy!(cp::ConsumerProblem, V::Matrix, out::Matrix)
    bellman_operator!(cp, V, out, ret_policy=true)
end

function get_greedy(cp::ConsumerProblem, V::Matrix)
    bellman_operator(cp, V, ret_policy=true)
end

"""

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.

##### Arguments

- `cp::CareerWorkerProblem` : Instance of `CareerWorkerProblem`
- `c::Matrix` : Current guess for the policy function
- `out::Matrix` : Storage for output

##### Returns

None, `out` is updated in place to hold the policy function

"""

function coleman_operator!(cp::ConsumerProblem, c::Matrix, out::Matrix)
    # simplify names, set up arrays
    R, Pi, bet, du, b = cp.R, cp.Pi, cp.bet, cp.du, cp.b
    asset_grid, z_vals = cp.asset_grid, cp.z_vals
    z_size = length(z_vals)
    gam = R * bet
    vals = Array(Float64, z_size)

    # linear interpolation to get consumption function. Updates vals inplace

```

```

function cf!(a, vals)
    for i=1:z_size
        vals[i] = CoordInterpGrid(asset_grid, c[:, i], BCnearest,
                                  InterpLinear)[a]
    end
    nothing
end

# compute lower_bound for optimization
opt_lb = minimum(z_vals) - 1e-2
for (i_z, z) in enumerate(z_vals)
    for (i_a, a) in enumerate(asset_grid)
        function h(t)
            cf!(R*a+z-t, vals) # update vals
            expectation = dot(du(vals), vec(Pi[i_z, :]))
            return abs(du(t) - max(gam * expectation, du(R*a+z+b)))
        end
        opt_ub = R*a + z + b # addresses issue #8 on github
        res = optimize(h, min(opt_lb, opt_ub - 1e-2), opt_ub, method=:brent)
        out[i_a, i_z] = res.minimum
    end
end
return out
end

"""
Apply the Coleman operator for a given model and initial value

See the specific methods of the mutating version of this function for more
details on arguments
"""
function coleman_operator(cp::ConsumerProblem, c::Matrix)
    out = similar(c)
    coleman_operator!(cp, c, out)
    return out
end

function init_values(cp::ConsumerProblem)
    # simplify names, set up arrays
    R, bet, u, b = cp.R, cp.bet, cp.u, cp.b
    asset_grid, z_vals = cp.asset_grid, cp.z_vals
    shape = length(asset_grid), length(z_vals)
    V, c = Array(Float64, shape...), Array(Float64, shape...)

    # Populate V and c
    for (i_z, z) in enumerate(z_vals)
        for (i_a, a) in enumerate(asset_grid)
            c_max = R*a + z + b
            c[i_a, i_z] = c_max
            V[i_a, i_z] = u(c_max) ./ (1 - bet)
        end
    end
end

```

```

    end

    return V, c
end

```

The code contains a type 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

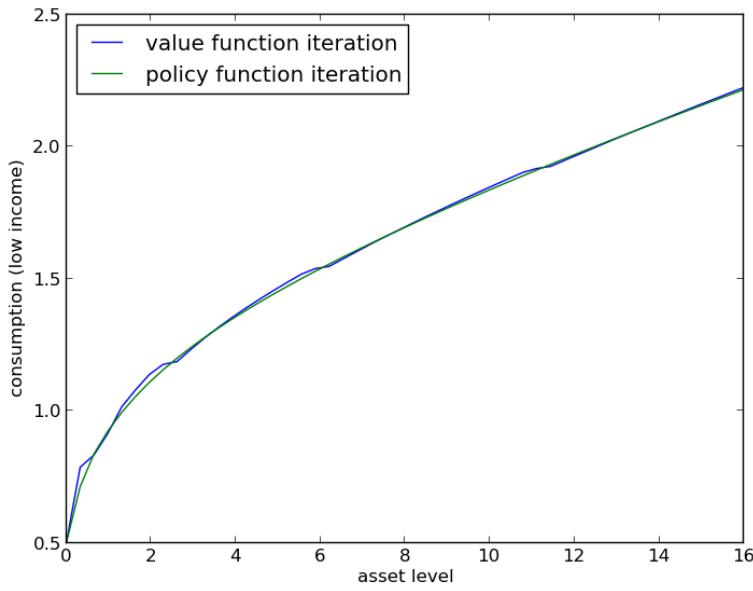
Consumption is shown as a function of assets with income  $z$  held fixed at its smallest value

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(cp)`
- 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 Julia console, a comparison of the operators can be made as follows



```
julia> using QuantEcon

julia> cp = ConsumerProblem();

julia> v, c, = initialize(cp);

julia> @time bellman_operator(cp, v);
elapsed time: 0.095017748 seconds (24212168 bytes allocated, 30.48% gc time)

julia> @time coleman_operator(cp, c);
elapsed time: 0.0696242 seconds (23937576 bytes allocated)
```

**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

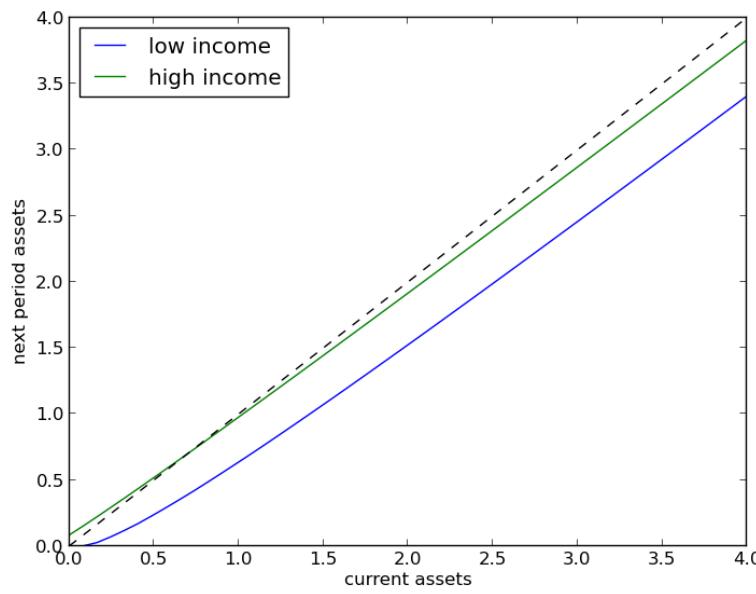
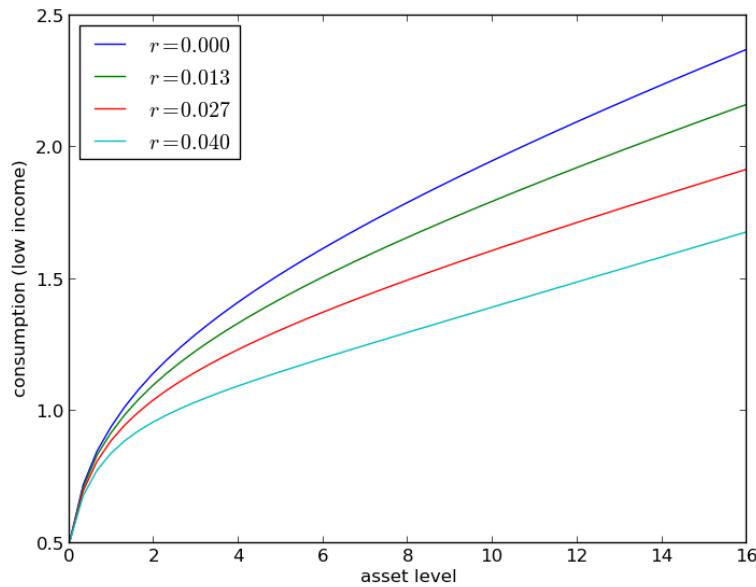
- Other than  $r$ , all parameters are at their default values
- $r$  steps through  $linspace(0, 0.04, 4)$
- Consumption is plotted against assets for income shock fixed at the smallest value

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

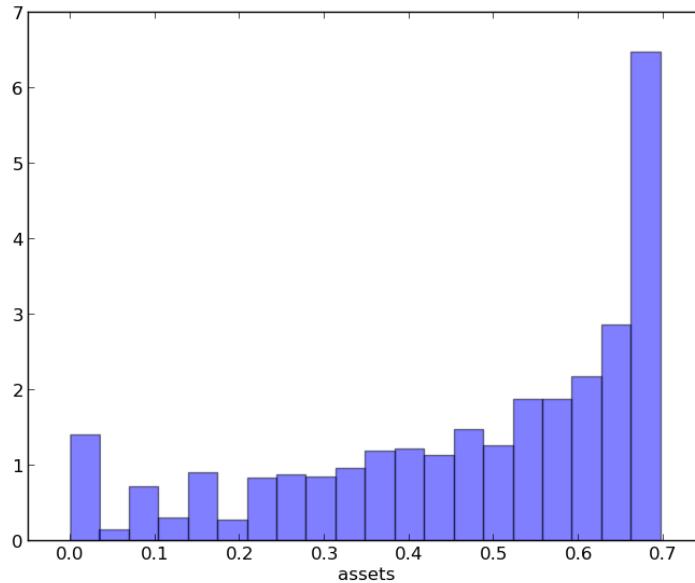
We can see from the figure that the dynamics will be stable — assets do not diverge

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 function `mc_sample_path` from `quantecon`

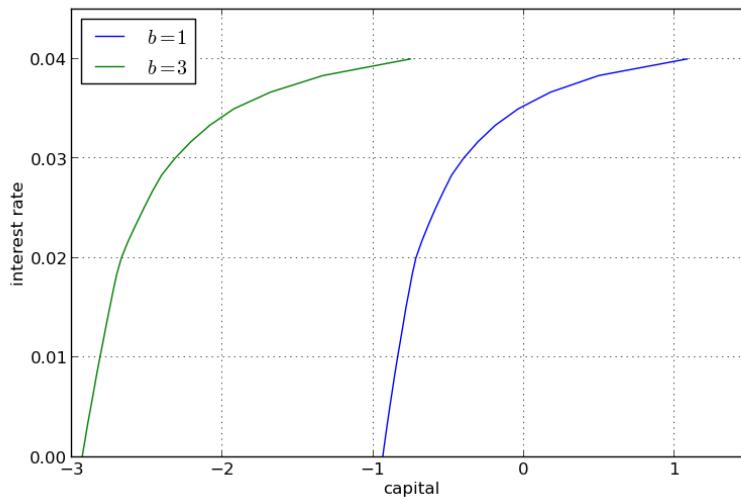
**Exercise 4** Following on from exercises 2 and 3, let's look at how savings and aggregate asset holdings vary with the interest rate

- 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](#)

## 3.7 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.

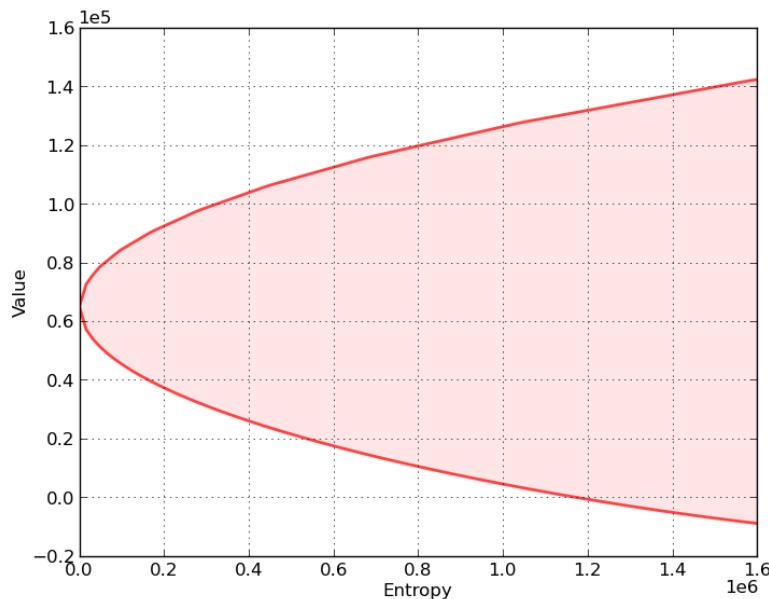
---

**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

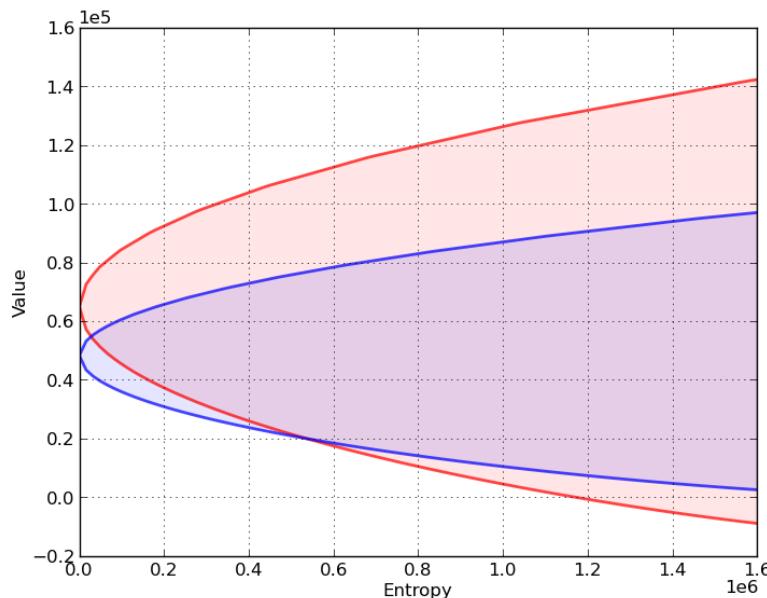
- Entropy is zero when the set includes only the approximating model, indicating that the decision maker completely trusts the approximating model
- Entropy is bigger, and the set of surrounding models is bigger, the less the decision maker trusts the approximating model

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

$$\sum_{t=0}^{\infty} \beta^t \{x'_t R x_t + u'_t Q u_t\} \quad (3.54)$$

subject to the linear law of motion

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \quad t = 0, 1, 2, \dots \quad (3.55)$$

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

$$J(x) = \min_u \max_w \{x'Rx + u'Qu + \beta [J(Ax + Bu + Cw) - \theta w'w]\} \quad (3.56)$$

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 (3.56) 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

$$\begin{aligned} \max_w \{(Ax + Bu + Cw)'P(Ax + Bu + Cw) - \theta w'w\} \\ = (Ax + Bu)'D(P)(Ax + Bu) \end{aligned} \quad (3.57)$$

where

$$D(P) := P + PC(\theta I - C'PC)^{-1}C'P \quad (3.58)$$

and  $I$  is a  $j \times j$  identity matrix. Substituting this expression for the maximum into (3.56) yields

$$x'Px = \min_u \{x'Rx + u'Qu + \beta(Ax + Bu)'D(P)(Ax + Bu)\} \quad (3.59)$$

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 (3.59) 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 (3.58) 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

$$\hat{F} := (Q + \beta B'D(\hat{P})B)^{-1}\beta B'D(\hat{P})A \quad (3.60)$$

We also define

$$\hat{K} := (\theta I - C'\hat{P}C)^{-1}C'\hat{P}(A - B\hat{F}) \quad (3.61)$$

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 (3.57) evaluated at the fixed rule  $u = -\hat{F}x$

Note that  $\hat{P}, \hat{F}, \hat{K}$  are all determined by the primitives and  $\theta$

Note also that if  $\theta$  is very large, then  $D$  is approximately equal to the identity mapping

Hence, when  $\theta$  is large,  $\hat{P}$  and  $\hat{F}$  are approximately equal to their standard LQ values

Furthermore, when  $\theta$  is large,  $\hat{K}$  is approximately equal to zero

Conversely, smaller  $\theta$  is associated with greater fear of model misspecification, and greater concern for robustness

### 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

$$\sum_{t=1}^{\infty} \beta^t w_t' w_t \leq \eta \quad (3.62)$$

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 (3.54), this value can be written as

$$J_F(x_0, \mathbf{w}) := \sum_{t=0}^{\infty} \beta^t x_t' (R + F' QF) x_t \quad (3.63)$$

where

$$x_{t+1} = (A - BF)x_t + Cw_{t+1} \quad (3.64)$$

and the initial condition  $x_0$  is as specified in the left side of (3.63)

Agent 2 chooses  $\mathbf{w}$  to maximize agent 1's loss  $J_F(x_0, \mathbf{w})$  subject to (3.62)

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 (3.64) and  $\theta$  is a Lagrange multiplier on constraint (3.62)

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,

$$\min_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \{ -x_t' (R + F' QF) x_t + \beta\theta w_{t+1}' w_{t+1} \} \quad (3.65)$$

subject to (3.64)

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 (3.62), which can be done by choosing  $\theta = \theta_\eta$  such that

$$\beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \theta_\eta)' K(F, \theta_\eta) x_t = \eta \quad (3.66)$$

Here  $x_t$  is given by (3.64) — 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 (3.65) 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

$$R_\theta(x_0, F) - \theta \text{ent} \leq \sum_{t=0}^{\infty} \beta^t \{ -x_t' (R + F' QF) x_t \} \quad (3.67)$$

where

$$\text{ent} := \beta \sum_{t=0}^{\infty} \beta^t w_{t+1}' w_{t+1}$$

The left side of inequality (3.67) 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 (3.67) is attained when

$$\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \theta)' K(F, \theta) x_t \quad (3.68)$$

To construct the *lower bound* on the set of values associated with all perturbations  $w$  satisfying the entropy constraint (3.62) at a given entropy level, we proceed as follows:

- For a given  $\theta$ , solve the minimization problem (3.65)
- Compute the minimizer  $R_\theta(x_0, F)$  and the associated entropy using (3.68)
- 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 (3.65) with the *maximization* problem

$$V_{\tilde{\theta}}(x_0, F) = \max_{\mathbf{w}} \sum_{t=0}^{\infty} \beta^t \left\{ -x'_t(R + F'QF)x_t - \beta\tilde{\theta}w'_{t+1}w_{t+1} \right\} \quad (3.69)$$

where now  $\tilde{\theta} > 0$  penalizes the choice of  $\mathbf{w}$  with larger entropy.

(Notice that  $\tilde{\theta} = -\theta$  in problem (3.65))

Because “maximizers maximize” we have

$$V_{\tilde{\theta}}(x_0, F) \geq \sum_{t=0}^{\infty} \beta^t \left\{ -x'_t(R + F'QF)x_t \right\} - \beta\tilde{\theta} \sum_{t=0}^{\infty} \beta^t w'_{t+1}w_{t+1}$$

which in turn implies the inequality

$$V_{\tilde{\theta}}(x_0, F) + \tilde{\theta} \text{ent} \geq \sum_{t=0}^{\infty} \beta^t \left\{ -x'_t(R + F'QF)x_t \right\} \quad (3.70)$$

where

$$\text{ent} \equiv \beta \sum_{t=0}^{\infty} \beta^t w'_{t+1}w_{t+1}$$

The left side of inequality (3.70) is a straight line with slope  $\tilde{\theta}$

The upper bound on the left side of (3.70) is attained when

$$\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x'_t K(F, \tilde{\theta})' K(F, \tilde{\theta}) x_t \quad (3.71)$$

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 (3.69)
- Compute the maximizer  $V_{\tilde{\theta}}(x_0, F)$  and the associated entropy using (3.71)
- Compute the upper bound on the value function  $V_{\tilde{\theta}}(x_0, F) + \tilde{\theta} \text{ent}$  and plot it against ent
- Repeat the preceding three steps for a range of values of  $\tilde{\theta}$  to trace out the upper bound

**Reshaping the set of values** Now in the interest of *reshaping* these sets of values by choosing  $F$ , we turn to agent 1’s problem

**Agent 1’s Problem** Now we turn to agent 1, who solves

$$\min_{\{u_t\}} \sum_{t=0}^{\infty} \beta^t \left\{ x'_t R x_t + u'_t Q u_t - \beta\theta w'_{t+1} w_{t+1} \right\} \quad (3.72)$$

where  $\{w_{t+1}\}$  satisfies  $w_{t+1} = Kx_t$

In other words, agent 1 minimizes

$$\sum_{t=0}^{\infty} \beta^t \{x_t'(R - \beta\theta K'K)x_t + u_t'Qu_t\} \quad (3.73)$$

subject to

$$x_{t+1} = (A + CK)x_t + Bu_t \quad (3.74)$$

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 (3.60) 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 (3.61)
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 (3.56) with the stochastic analogue

$$J(x) = \min_u \max_{\psi \in \mathcal{P}} \left\{ x' Rx + u' Qu + \beta \left[ \int J(Ax + Bu + Cw) \psi(dw) - \theta D_{KL}(\psi, \phi) \right] \right\} \quad (3.75)$$

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 (3.56), since it discourages large deviations from the benchmark

**Solving the Model** The maximization problem in (3.75) 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  $(I - \theta^{-1} C' P C)^{-1}$  is nonsingular, then

$$\begin{aligned} \max_{\psi \in \mathcal{P}} & \left\{ \int (Ax + Bu + Cw)' P (Ax + Bu + Cw) \psi(dw) - \theta D_{KL}(\psi, \phi) \right\} \\ &= (Ax + Bu)' \mathcal{D}(P)(Ax + Bu) + \kappa(\theta, P) \end{aligned} \quad (3.76)$$

where

$$\kappa(\theta, P) := \theta \ln[\det(I - \theta^{-1} C' P C)^{-1}]$$

and the maximizer is the Gaussian distribution

$$\psi = N \left( (\theta I - C' P C)^{-1} C' P (Ax + Bu), (I - \theta^{-1} C' P C)^{-1} \right) \quad (3.77)$$

Substituting the expression for the maximum into Bellman equation (3.75) and using  $J(x) = x' Px + d$  gives

$$x' Px + d = \min_u \left\{ x' Rx + u' Qu + \beta (Ax + Bu)' \mathcal{D}(P)(Ax + Bu) + \beta [d + \kappa(\theta, P)] \right\} \quad (3.78)$$

Since constant terms do not affect minimizers, the solution is the same as (3.59), leading to

$$x' Px + d = x' \mathcal{B}(\mathcal{D}(P))x + \beta [d + \kappa(\theta, P)]$$

To solve this Bellman equation, we take  $\hat{P}$  to be the positive definite fixed point of  $\mathcal{B} \circ \mathcal{D}$

In addition, we take  $\hat{d}$  as the real number solving  $d = \beta [d + \kappa(\theta, P)]$ , which is

$$\hat{d} := \frac{\beta}{1 - \beta} \kappa(\theta, P) \quad (3.79)$$

The robust policy in this stochastic case is the minimizer in (3.78), which is once again  $u = -\hat{F}x$  for  $\hat{F}$  given by (3.60)

Substituting the robust policy into (3.77) 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 (3.61)

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 (3.75), 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 (3.76) 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

$$d_F := \frac{\beta}{1 - \beta} \kappa(\theta, P_F) = \frac{\beta}{1 - \beta} \theta \ln[\det(I - \theta^{-1}C'P_FC)^{-1}] \quad (3.80)$$

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` package provides a type called `RBLQ` for implementation of robust LQ optimal control

Here's the relevant code, from file `robustlq.jl` “

```

#=

Provides a type called RBLQ for solving robust linear quadratic control
problems.

@author : Spencer Lyon <spencer.lyon@nyu.edu>

@date : 2014-08-19

References
-----
http://quant-econ.net/jl/robustness.html
=#

"""

Represents infinite horizon robust LQ control problems of the form

    min_{u_t} sum_t beta^t {x_t' R x_t + u_t' Q u_t }

subject to

    x_{t+1} = A x_t + B u_t + C w_{t+1}

and with model misspecification parameter theta.

##### Fields

- `Q::Matrix{Float64}` : The cost(payoff) matrix for the controls. See above
for more. `Q` should be k x k and symmetric and positive definite
- `R::Matrix{Float64}` : The cost(payoff) matrix for the state. See above for
more. `R` should be n x n and symmetric and non-negative definite
- `A::Matrix{Float64}` : The matrix that corresponds with the state in the
state space system. `A` should be n x n
- `B::Matrix{Float64}` : The matrix that corresponds with the control in the
state space system. `B` should be n x k
- `C::Matrix{Float64}` : The matrix that corresponds with the random process in
the state space system. `C` should be n x j
- `beta::Real` : The discount factor in the robust control problem
- `theta::Real` : The robustness factor in the robust control problem
- `k, n, j::Int` : Dimensions of input matrices

"""

type RBLQ
    A::Matrix
    B::Matrix
    C::Matrix
    Q::Matrix
    R::Matrix
    k::Int
    n::Int
    j::Int
    bet::Real

```

```

    theta::Real
end

function RBLQ(Q::ScalarOrArray, R::ScalarOrArray, A::ScalarOrArray,
              B::ScalarOrArray, C::ScalarOrArray, bet::Real, theta::Real)
    k = size(Q, 1)
    n = size(R, 1)
    j = size(C, 2)

    # coerce sizes
    A = reshape([A;], n, n)
    B = reshape([B;], n, k)
    C = reshape([C;], n, j)
    R = reshape([R;], n, n)
    Q = reshape([Q;], k, k)
    RBLQ(A, B, C, Q, R, k, n, j, bet, theta)
end

"""
The D operator, mapping P into

D(P) := P + PC(theta I - C'PC)^{-1} C'P.

##### Arguments

- `rlq::RBLQ`: Instance of `RBLQ` type
- `P::Matrix{Float64}` : `size` is n x n

##### Returns

- `dP::Matrix{Float64}` : The matrix P after applying the D operator

"""
function d_operator(rlq::RBLQ, P::Matrix)
    C, theta, I = rlq.C, rlq.theta, eye(rlq.j)
    S1 = P*C
    dP = P + S1*((theta.*I - C'*S1) \ (S1'))

    return dP
end

"""
The D operator, mapping P into

B(P) := R - beta^2 A'PB(Q + beta B'PB)^{-1}B'PA + beta A'PA

and also returning

F := (Q + beta B'PB)^{-1} beta B'PA

#####
Arguments

```

```

- `rlq::RBLQ` : Instance of `RBLQ` type
- `P::Matrix{Float64}` : `size` is n x n

##### Returns

- `F::Matrix{Float64}` : The F matrix as defined above
- `new_p::Matrix{Float64}` : The matrix P after applying the B operator

"""
function b_operator(rlq::RBLQ, P::Matrix)
    A, B, Q, R, bet = rlq.A, rlq.B, rlq.Q, rlq.R, rlq.bet

    S1 = Q + bet.*B'*P*B
    S2 = bet.*B'*P*A
    S3 = bet.*A'*P*A

    F = S1 \ S2
    new_P = R - S2'*F + S3

    return F, new_P
end

"""
Solves the robust control problem.

```

The algorithm here tricks the problem into a stacked LQ problem, as described in chapter 2 of Hansen- Sargent's text "Robustness." The optimal control with observed state is

$$u_t = -F x_t$$

And the value function is  $-x'Px$

```

##### Arguments

- `rlq::RBLQ` : Instance of `RBLQ` type

##### Returns

- `F::Matrix{Float64}` : The optimal control matrix from above
- `P::Matrix{Float64}` : The positive semi-definite matrix defining the value
  function
- `K::Matrix{Float64}` : the worst-case shock matrix `K`, where
  `w_{t+1} = K x_t` is the worst case shock

"""
function robust_rule(rlq::RBLQ)
    A, B, C, Q, R = rlq.A, rlq.B, rlq.C, rlq.Q, rlq.R
    bet, theta, k, j = rlq.bet, rlq.theta, rlq.k, rlq.j

    # Set up LQ version
    I = eye(j)

```

```

Z = zeros(k, j)
Ba = [B C]
Qa = [Q Z
      Z' -bet.*I.*theta]
lq = LQ(Qa, R, A, Ba, bet=bet)

# Solve and convert back to robust problem
P, f, d = stationary_values(lq)
F = f[1:k, :]
K = -f[k+1:end, :]

return F, K, P
end

"""

Solve the robust LQ problem

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.

##### Arguments

- `rlq::RBLQ`: Instance of `RBLQ` type
- `P_init::Matrix{Float64}(zeros(rlq.n, rlq.n))` : The initial guess for the
value function matrix
- `;max_iter::Int(80)` : Maximum number of iterations that are allowed
- `;tol::Real(1e-8)` The tolerance for convergence

##### Returns

- `F::Matrix{Float64}` : The optimal control matrix from above
- `P::Matrix{Float64}` : The positive semi-definite matrix defining the value
function
- `K::Matrix{Float64}` : the worst-case shock matrix `K`, where
`w_{t+1} = K x_t` is the worst case shock

"""

function robust_rule_simple(rlq::RBLQ,
                           P::Matrix{Float64}=zeros(Float64, rlq.n, rlq.n);
                           max_iter=80,
                           tol=1e-8)
    # Simplify notation
    A, B, C, Q, R = rlq.A, rlq.B, rlq.C, rlq.Q, rlq.R
    bet, theta, k, j = rlq.bet, rlq.theta, rlq.k, rlq.j
    iterate, e = 0, tol + 1.0

    F = similar(P) # instantiate so available after loop

```

```

while iterate <= max_iter && e > tol
    F, new_P = b_operator(rlq, d_operator(rlq, P))
    e = sqrt(sum((new_P - P).^2))
    iterate += 1
    copy!(P, new_P)
end

if iterate >= max_iter
    warn("Maximum iterations in robust_rul_simple")
end

I = eye(j)
K = (theta.*I - C'*P*C)\(C'*P)*(A - B*F)

return F, K, P
end

"""
Compute agent 2's best cost-minimizing response `K`, given `F`.

##### Arguments

- `rlq::RBLQ`: Instance of `RBLQ` type
- `F::Matrix{Float64}`: A k x n array representing agent 1's policy

##### Returns

- `K::Matrix{Float64}` : Agent's best cost minimizing response corresponding to `F`
- `P::Matrix{Float64}` : The value function corresponding to `F`

"""

function F_to_K(rlq::RBLQ, F::Matrix)
    # simplify notation
    R, Q, A, B, C = rlq.R, rlq.Q, rlq.A, rlq.B, rlq.C
    bet, theta = rlq.bet, rlq.theta

    # set up lq
    Q2 = bet * theta
    R2 = - R - F'*Q*F
    A2 = A - B*F
    B2 = C
    lq = LQ(Q2, R2, A2, B2, bet=bet)

    neg_P, neg_K, d = stationary_values(lq)

    return -neg_K, -neg_P
end

"""
Compute agent 1's best cost-minimizing response `K`, given `F`.

##### Arguments

```

```

- `rlq::RBLQ`: Instance of `RBLQ` type
- `K::Matrix{Float64}`: A k x n array representing the worst case matrix

##### Returns

- `F::Matrix{Float64}` : Agent's best cost minimizing response corresponding to
`K`
- `P::Matrix{Float64}` : The value function corresponding to `K`

"""
function K_to_F(rlq::RBLQ, K::Matrix)
    R, Q, A, B, C = rlq.R, rlq.Q, rlq.A, rlq.B, rlq.C
    bet, theta = rlq.bet, rlq.theta

    A1, B1, Q1, R1 = A+C*K, B, Q, R-bet*theta.*K'*K
    lq = LQ(Q1, R1, A1, B1, bet=bet)

    P, F, d = stationary_values(lq)

    return F, P
end

"""
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.

##### Arguments

- `rlq::RBLQ`: Instance of `RBLQ` type
- `F::Matrix{Float64}` The policy function, a k x n array
- `K::Matrix{Float64}` The worst case matrix, a j x n array
- `x0::Vector{Float64}` : The initial condition for state

##### Returns

- `e::Float64` The deterministic entropy

"""
function compute_deterministic_entropy(rlq::RBLQ, F, K, x0)
    B, C, bet = rlq.B, rlq.C, rlq.bet
    H0 = K'*K
    C0 = zeros(Float64, rlq.n, 1)
    A0 = A - B*F + C*K
    return var_quadratic_sum(A0, C0, H0, bet, x0)
end

"""
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.

##### Arguments

```

```

- `rlq::RBLQ` : Instance of `RBLQ` type
- `F::Matrix{Float64}` : The policy function, a k x n array

##### Returns

- `P_F::Matrix{Float64}` : Matrix for discounted cost
- `d_F::Float64` : Constant for discounted cost
- `K_F::Matrix{Float64}` : Worst case policy
- `O_F::Matrix{Float64}` : Matrix for discounted entropy
- `o_F::Float64` : Constant for discounted entropy

"""
function evaluate_F(rlq::RBLQ, F::Matrix)
    R, Q, A, B, C = rlq.R, rlq.Q, rlq.A, rlq.B, rlq.C
    bet, theta, j = rlq.bet, rlq.theta, rlq.j

    # Solve for policies and costs using agent 2's problem
    K_F, P_F = F_to_K(rlq, F)
    I = eye(j)
    H = inv(I - C'*P_F*C./theta)
    d_F = log(det(H))

    # compute O_F and o_F
    sig = -1.0 / theta
    AO = sqrt(bet) .* (A - B*F + C*K_F)
    O_F = solve_discrete_lyapunov(AO', bet*K_F'*K_F)
    ho = (trace(H - 1) - d_F) / 2.0
    tr = trace(O_F*C*H*C')
    o_F = (ho + bet*tr) / (1 - bet)

    return K_F, P_F, d_F, O_F, o_F
end

```

Here is a brief description of the methods of the type

- `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 (3.60) – (3.61) 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 (3.66)
- `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 [examples/robust\\_monopolist.jl](#)

We repeat it here for convenience

```
#=
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 - gam (y_{t+1} - y_t)^2 / 2 - c y_t

The objective of the firm is  $E_t \sum_{t=0}^{\infty} \beta^t r_t$ 

For the linear regulator, we take the state and control to be

x_t = (1, y_t, d_t) and u_t = y_{t+1} - y_t

@author : Spencer Lyon <spencer.lyon@nyu.edu>

@date : 2014-07-05

References
-----
Simple port of the file examples/robust_monopolist.py

http://quant-econ.net/robustness.html#application

=#
using QuantEcon
using PyPlot
using Grid

# model parameters
a_0      = 100
a_1      = 0.5
rho      = 0.9
sigma_d = 0.05
bet      = 0.95
c        = 2
gam     = 50.0
theta   = 0.002
ac      = (a_0 - c) / 2.0

# Define LQ matrices
R = [0 ac 0
      ac -a_1 0.5
      0. 0.5 0]
R = -R # For minimization
Q = [gam / 2.0]'
```

```

    0. 0. rho]
B = [0. 1. 0.]'
C = [0. 0. sigma_d]'

## Functions

function evaluate_policy(theta, F)
    rlq = RBLQ(Q, R, A, B, C, bet, theta)
    K_F, P_F, d_F, O_F, o_F = evaluate_F(rlq, F)
    x0 = [1.0 0.0 0.0]'
    value = - x0'*P_F*x0 - d_F
    entropy = x0'*O_F*x0 + o_F
    return value[1], entropy[1] # return scalars
end

function value_and_entropy(emax, F, bw, grid_size=1000)
    if lowercase(bw) == "worst"
        thetas = 1 ./ linspace(1e-8, 1000, grid_size)
    else
        thetas = -1 ./ linspace(1e-8, 1000, grid_size)
    end

    data = Array(Float64, grid_size, 2)

    for (i, theta) in enumerate(thetas)
        data[i, :] = collect(evaluate_policy(theta, F))
        if data[i, 2] >= emax # stop at this entropy level
            data = data[1:i, :]
            break
        end
    end
    return data
end

## Main

# compute optimal rule
optimal_lq = LQ(Q, R, A, B, C, bet)
Po, Fo, Do = stationary_values(optimal_lq)

# compute robust rule for our theta
baseline_robust = RBLQ(Q, R, A, B, C, bet, theta)
Fb, Kb, Pb = robust_rule(baseline_robust)

# Check the positive definiteness of worst-case covariance matrix to
# ensure that theta exceeds the breakdown point
test_matrix = eye(size(Pb, 1)) - (C' * Pb * C ./ theta)[1]
eigenvals, eigenvecs = eig(test_matrix)
@assert all(eigenvals .>= 0)

emax = 1.6e6

```

```

# compute values and entropies
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")

# plot results
fig, ax = 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))
end

plot_args = {:lw => 2, :alpha => 0.7}
colors = ("r", "b")

# we reverse order of "worst_case"s so values are ascending
data_pairs = ((optimal_best_case, optimal_worst_case),
               (robust_best_case, robust_worst_case))

egrid = linspace(0, emax, 100)
egrid_data = Array{Float64}[]
for (c, data_pair) in zip(colors, data_pairs)
    for data in data_pair
        x, y = data[:, 2], data[:, 1]
        curve(z) = InterpIrregular(x, y, BCnearest, InterpLinear)[z]
        ax[:plot](egrid, curve(egrid), color=c; plot_args...)
        push!(egrid_data, curve(egrid))
    end
end
ax[:fill_between](egrid, egrid_data[1], egrid_data[2],
                  color=colors[1], alpha=0.1)
ax[:fill_between](egrid, egrid_data[3], egrid_data[4],
                  color=colors[2], alpha=0.1)
plt.show()

```

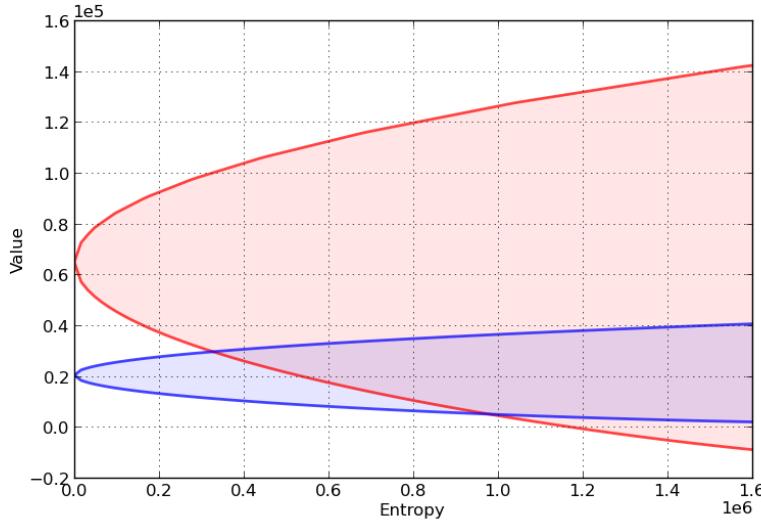
Here's another such figure, with  $\theta = 0.002$  instead of 0.02

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 (3.61)

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 (3.60), then

$$K(\hat{F}, \theta) = (\theta I - C' \hat{P} C)^{-1} C' \hat{P} (A - B \hat{F}) \quad (3.81)$$

*Proof:* As a first step, observe that when  $F = \hat{F}$ , the Bellman equation associated with the LQ problem (3.64) – (3.65) is

$$\tilde{P} = -R - \hat{F}' Q \hat{F} - \beta^2 (A - B \hat{F})' \tilde{P} C (\beta \theta I + \beta C' \tilde{P} C)^{-1} C' \tilde{P} (A - B \hat{F}) + \beta (A - B \hat{F})' \tilde{P} (A - B \hat{F}) \quad (3.82)$$

(revisit [this discussion](#) if you don't know where (3.82) 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 (3.82)

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 (3.81)

Hence it remains only to show that  $-\hat{P}$  solves (3.82), 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  $\hat{P} = \mathcal{B}(\mathcal{D}(\hat{P}))$ )

## 3.8 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*

$$X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}, \quad t \in \mathbb{Z} \quad (3.83)$$

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 (3.83) is

$$\gamma(k) = \sigma^2 \sum_{j=0}^{\infty} \psi_j \psi_{j+k} \quad (3.84)$$

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

$$X_t = \phi X_{t-1} + \epsilon_t \quad \text{where } |\phi| < 1 \quad \text{and } \{\epsilon_t\} \text{ is white noise} \quad (3.85)$$

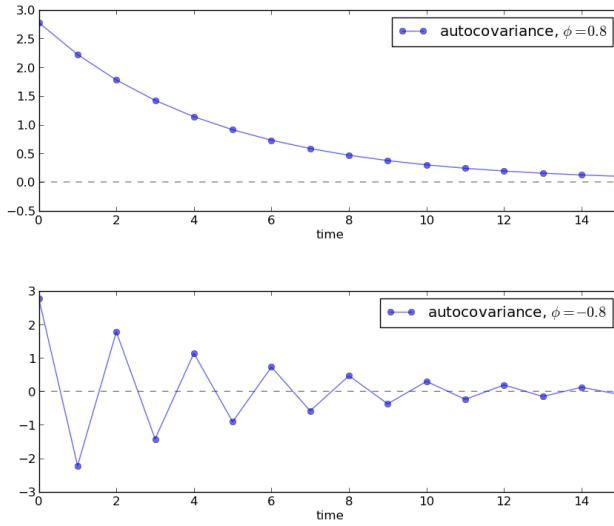
By direct substitution, it is easy to verify that  $X_t = \sum_{j=0}^{\infty} \phi^j \epsilon_{t-j}$

Hence  $\{X_t\}$  is a general linear process

Applying (3.84) to the previous expression for  $X_t$ , we get the AR(1) autocovariance function

$$\gamma(k) = \phi^k \frac{\sigma^2}{1 - \phi^2}, \quad k = 0, 1, \dots \quad (3.86)$$

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

$$X_t = \phi_1 X_{t-1} + \dots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \dots + \theta_q \epsilon_{t-q} \quad (3.87)$$

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 (3.87) as

$$L^0 X_t - \phi_1 L^1 X_t - \cdots - \phi_p L^p X_t = L^0 \epsilon_t + \theta_1 L^1 \epsilon_t + \cdots + \theta_q L^q \epsilon_t \quad (3.88)$$

If we let  $\phi(z)$  and  $\theta(z)$  be the polynomials

$$\phi(z) := 1 - \phi_1 z - \cdots - \phi_p z^p \quad \text{and} \quad \theta(z) := 1 + \theta_1 z + \cdots + \theta_q z^q \quad (3.89)$$

then (3.88) simplifies further to

$$\phi(L) X_t = \theta(L) \epsilon_t \quad (3.90)$$

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

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k) \tag{3.91}$$

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 :index'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

$$f(\omega) = \sigma^2(1 + 2\theta \cos(\omega) + \theta^2) \quad (3.92)$$

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

$$f(\omega) = \frac{\sigma^2}{1 - 2\phi \cos(\omega) + \phi^2} \quad (3.93)$$

More generally, it can be shown that the spectral density of the ARMA process (3.87) is

$$f(\omega) = \left| \frac{\theta(e^{i\omega})}{\phi(e^{i\omega})} \right|^2 \sigma^2 \quad (3.94)$$

where

- $\sigma$  is the standard deviation of the white noise process  $\{\epsilon_t\}$
- the polynomials  $\phi(\cdot)$  and  $\theta(\cdot)$  are as defined in (3.89)

The derivation of (3.94) 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 (3.92) and (3.93) are indeed special cases of (3.94)

**Interpreting the Spectral Density** Plotting (3.93) 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*

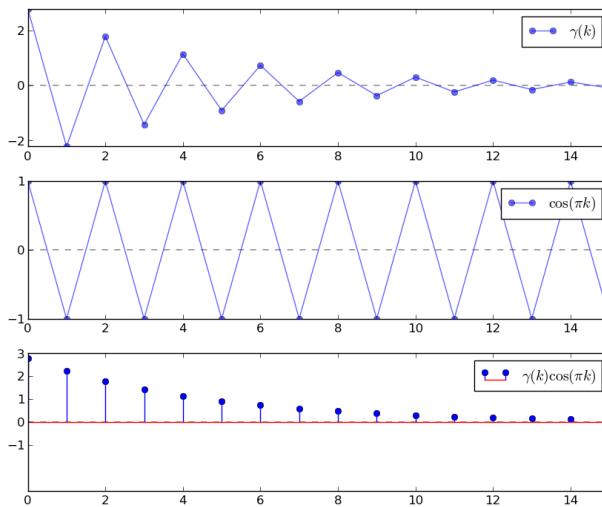
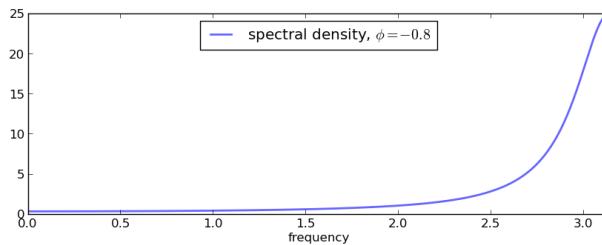
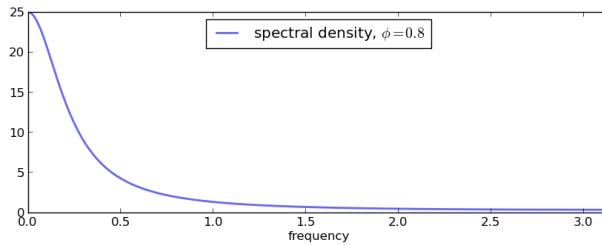
Informally, we think of the spectral density as being large at those  $\omega \in [0, \pi]$  such that the autocovariance function exhibits significant cycles at this “frequency”

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

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k) = \gamma(0) + 2 \sum_{k \geq 1} (-0.8)^k \cos(\omega k) \quad (3.95)$$

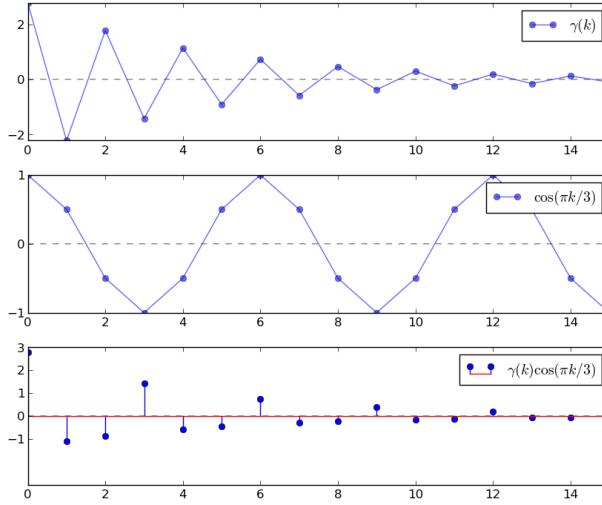
When we evaluate this at  $\omega = \pi$ , we get a large number because  $\cos(\pi k)$  is large and positive when  $(-0.8)^k$  is positive, and large in absolute value and negative when  $(-0.8)^k$  is negative



Hence the product is always large and positive, and hence the sum of the products on the right-hand side of (3.95) 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

$$\gamma(k) = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(\omega) e^{i\omega k} d\omega \quad (3.96)$$

This is convenient in situations where the spectral density is easier to calculate and manipulate than the autocovariance function

(For example, the expression (3.94) 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

$$f = \sum_k \alpha_k h_k \quad \text{where} \quad \alpha_k := \langle f, h_k \rangle \quad (3.97)$$

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

$$T\gamma = \sum_{k \in \mathbb{Z}} \gamma(k) \frac{e^{i\omega k}}{\sqrt{2\pi}} = \frac{1}{\sqrt{2\pi}} f(\omega) \quad (3.98)$$

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 (3.98) gives (3.96), justifying our earlier expression for the inverse transform

### Implementation

Most code for working with covariance stationary models deals with ARMA models

Julia code for studying ARMA models can be found in the `DSP.jl` package

Since this code doesn't quite cover our needs — particularly vis-a-vis spectral analysis — we've put together the module `arma.jl`, which is part of `QuantEcon` package.

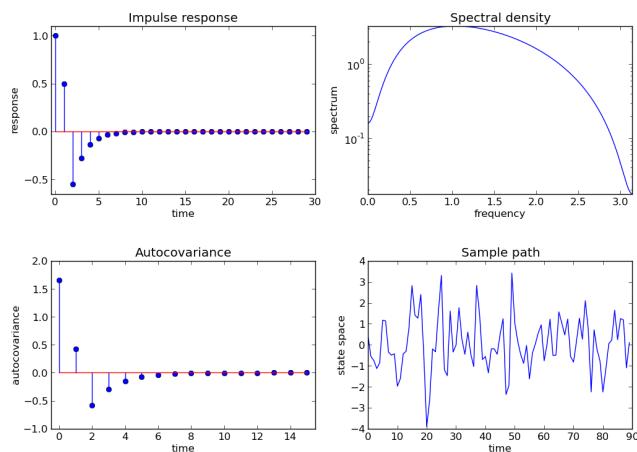
The module provides functions for mapping ARMA( $p, q$ ) models into their

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.jl" is printed below

```
#=
@authors: John Stachurski
Date: Thu Aug 21 11:09:30 EST 2014

Provides functions for working with and visualizing scalar ARMA processes.
Ported from Python module quantecon.arma, which was written by Doc-Jin Jang,
Jerry Choi, Thomas Sargent and John Stachurski

References
-----
http://quant-econ.net/jl/arma.html

=#
"""

Represents a scalar ARMA(p, q) process

If phi and theta are scalars, then the model is
understood to be

    X_t = phi X_{t-1} + epsilon_t + theta epsilon_{t-1}

where epsilon_t is a white noise process with standard
deviation sigma.

If phi and theta are arrays or sequences,
then the interpretation is the ARMA(p, q) model

    X_t = phi_1 X_{t-1} + ... + phi_p X_{t-p} +
          epsilon_t + theta_1 epsilon_{t-1} + ... +
          theta_q epsilon_{t-q}

where

* phi = (phi_1, phi_2, ..., phi_p)
* theta = (theta_1, theta_2, ..., theta_q)
* sigma is a scalar, the standard deviation of the white noise

##### Fields

- `phi::Vector` : AR parameters phi_1, ..., phi_p
- `theta::Vector` : MA parameters theta_1, ..., theta_q
- `p::Integer` : Number of AR coefficients
- `q::Integer` : Number of MA coefficients
- `sigma::Real` : Standard deviation of white noise
- `ma_poly::Vector` : MA polynomial --- filtering representatoin
- `ar_poly::Vector` : AR polynomial --- filtering representation

##### Examples
```

```

```julia
using QuantEcon
phi = 0.5
theta = [0.0, -0.8]
sigma = 1.0
lp = ARMA(phi, theta, sigma)
require(joinpath(Pkg.dir("QuantEcon"), "examples", "arma_plots.jl"))
quad_plot(lp)
```
"""

type ARMA
    phi::Vector      # AR parameters phi_1, ..., phi_p
    theta::Vector    # MA parameters theta_1, ..., theta_q
    p::Integer       # Number of AR coefficients
    q::Integer       # Number of MA coefficients
    sigma::Real      # Variance of white noise
    ma_poly::Vector  # MA polynomial --- filtering representation
    ar_poly::Vector  # AR polynomial --- filtering representation
end

# constructors to coerce phi/theta to vectors
ARMA(phi::Real, theta::Real=0.0, sigma::Real=1.0) = ARMA([phi], [theta], sigma)
ARMA(phi::Real, theta::Vector=[0.0], sigma::Real=1.0) = ARMA([phi], theta, sigma)
ARMA(phi::Vector, theta::Real=0.0, sigma::Real=1.0) = ARMA(phi, theta, sigma)

function ARMA(phi::AbstractVector, theta::AbstractVector=[0.0], sigma::Real=1.0)
    # == Record dimensions == #
    p = length(phi)
    q = length(theta)

    # == Build filtering representation of polynomials == #
    ma_poly = [1.0; theta]
    ar_poly = [1.0; -phi]
    return ARMA(phi, theta, p, q, sigma, ma_poly, ar_poly)
end

"""
Compute the spectral density function.

The spectral density is the discrete time Fourier transform of the
autocovariance function. In particular,

```

$$f(w) = \sum_k \gamma(k) \exp(-ikw)$$

where  $\gamma$  is the autocovariance function and the sum is over the set of all integers.

```

##### Arguments

- `arma::ARMA` : Instance of `ARMA` type
- `;two_pi::Bool(true)` : Compute the spectral density function over  $[0, \pi]$  if false and  $[0, 2\pi]$  otherwise.
- `;res(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::Vector{Float64}`: The normalized frequencies at which h was computed, in
  radians/sample
- `spect::Vector{Float64}` : The frequency response
"""

function spectral_density(arma::ARMA; res=1200, two_pi::Bool=true)
    # Compute the spectral density associated with ARMA process arma
    wmax = two_pi ? 2pi : pi
    w = linspace(0, wmax, res)
    tf = TFFilter(reverse(arma.ma_poly), reverse(arma.ar_poly))
    h = freqz(tf, w)
    spect = arma.sigma^2 * abs(h).^2
    return w, spect
end

"""

Compute the autocovariance function from the ARMA parameters
over the integers range(num_autocov) using the spectral density
and the inverse Fourier transform.

##### Arguments

- `arma::ARMA`: Instance of `ARMA` type
- `;num_autocov::Integer(16)` : The number of autocovariances to calculate

"""

function autocovariance(arma::ARMA; num_autocov::Integer=16)
    # Compute the autocovariance function associated with ARMA process arma
    # Computation is via the spectral density and inverse FFT
    (w, spect) = spectral_density(arma)
    acov = real(Base.ifft(spect))
    # num_autocov should be <= len(acov) / 2
    return acov[1:num_autocov]
end

"""

Get the impulse response corresponding to our model.

##### Arguments

- `arma::ARMA`: Instance of `ARMA` type
- `;impulse_length::Integer(30)` : Length of horizon for calucluating impulse
  reponse. Must be at least as long as the `p` fields of `arma`


##### Returns

- `psi::Vector{Float64}`: `psi[j]` is the response at lag j of the impulse
  response. We take psi[1] as unity.

```

```
"""
function impulse_response(arma::ARMA; impulse_length=30)
    # Compute the impulse response function associated with ARMA process arma
    err_msg = "Impulse length must be greater than number of AR coefficients"
    @assert impulse_length >= arma.p err_msg
    # == Pad theta with zeros at the end == #
    theta = [arma.theta; zeros(impulse_length - arma.q)]
    psi_zero = 1.0
    psi = Array(Float64, impulse_length)
    for j = 1:impulse_length
        psi[j] = theta[j]
        for i = 1:min(j, arma.p)
            psi[j] += arma.phi[i] * (j-i > 0 ? psi[j-i] : psi_zero)
        end
    end
    return [psi_zero; psi[1:end-1]]
end

"""

Compute a simulated sample path assuming Gaussian shocks.

##### Arguments

- `arma::ARMA`: Instance of `ARMA` type
- `ts_length::Integer(90)`: Length of simulation
- `;impulse_length::Integer(30)`: Horizon for calculating impulse response
  (see also docstring for `impulse_response`)

##### Returns

- `X::Vector{Float64}`: Simulation of the ARMA model `arma`

"""

function simulation(arma::ARMA; ts_length=90, impulse_length=30)
    # Simulate the ARMA process arma assuing Gaussian shocks
    J = impulse_length
    T = ts_length
    psi = impulse_response(arma, impulse_length=impulse_length)
    epsilon = arma.sigma * randn(T + J)
    X = Array(Float64, T)
    for t=1:T
        X[t] = dot(epsilon[t:J+t-1], psi)
    end
    return X
end
```

Here's an example of usage

```
julia> using QuantEcon

julia> using QuantEcon

julia> phi = 0.5;
```

```
julia> theta = [0, -0.8];
julia> lp = ARMA(phi, theta);
julia> QuantEcon.quad_plot(lp)
```

**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 `DSP.jl` and the `fft` routine in Julia

**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 Julia's Fourier transform routine `fft`, which wraps a standard C-based package called `FFTW`

A look at [the fft documentation](#) shows that the inverse transform `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 (3.96) we have now shown that, for  $n$  sufficiently large,  $a_k \approx \gamma(k)$  — which is exactly what we want to compute

## 3.9 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

$$I(\omega) := \frac{1}{n} \left| \sum_{t=0}^{n-1} X_t e^{it\omega} \right|^2, \quad \omega \in \mathbb{R} \quad (3.99)$$

(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 (3.99)

There are already functions available that will do this for us — an example is `periodogram` in the `DSP.jl` 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  $a_0, \dots, a_{n-1}$  stored in Julia array `a`, the function call `fft(a)` returns the values  $A_0, \dots, A_{n-1}$  as a Julia 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 `abs(fft(X)).^2 / length(X)`

Note: The Julia 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

```
function periodogram(x::Array):
    n = length(x)
    I_w = abs(fft(x)).^2 / n
    w = 2pi * [0:n-1] ./ n      # Fourier frequencies
    w, I_w = w[1:int(n/2)], I_w[1:int(n/2)]  # Truncate to interval [0, pi]
    return w, I_w
end
```

Let's generate some data for this function using the `ARMA` type 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

$$X_t = 0.5X_{t-1} + \epsilon_t - 0.8\epsilon_{t-2} \quad (3.100)$$

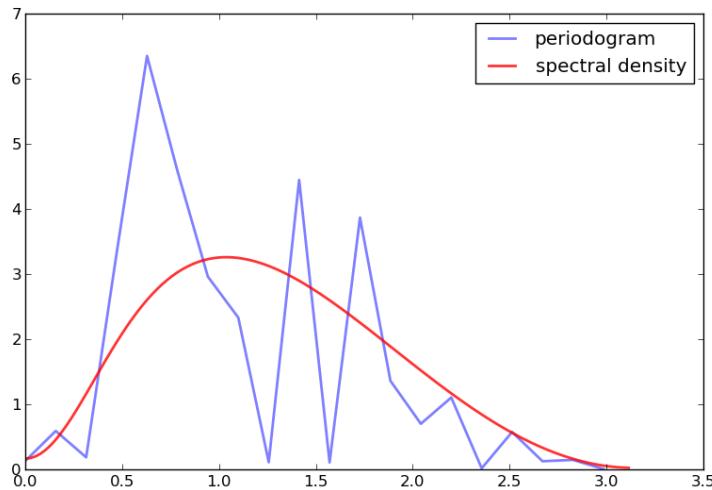
where  $\{\epsilon_t\}$  is white noise with unit variance, and compares the periodogram to the actual spectral density

```
import PyPlot: plt
import QuantEcon: ARMA

n = 40                      # Data size
phi, theta = 0.5, [0, -0.8]    # AR and MA parameters
lp = ARMA(phi, theta)
X = simulation(lp, 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 = spectral_density(lp, 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

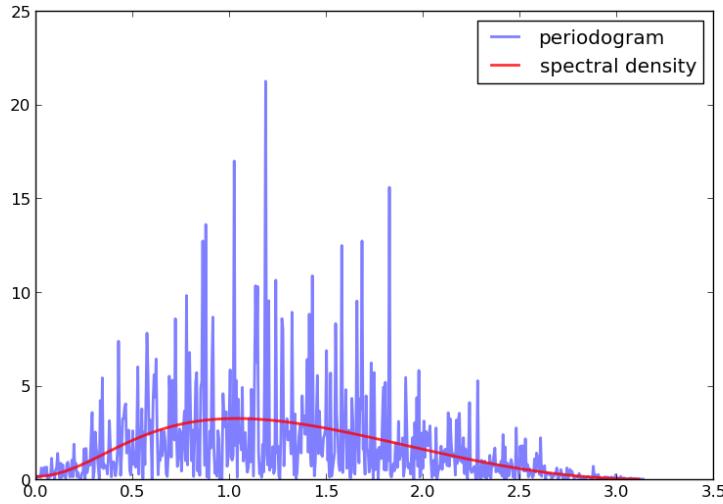


This estimate looks rather disappointing, but the data size is only 40, so perhaps it's not surprising that the estimate is poor

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

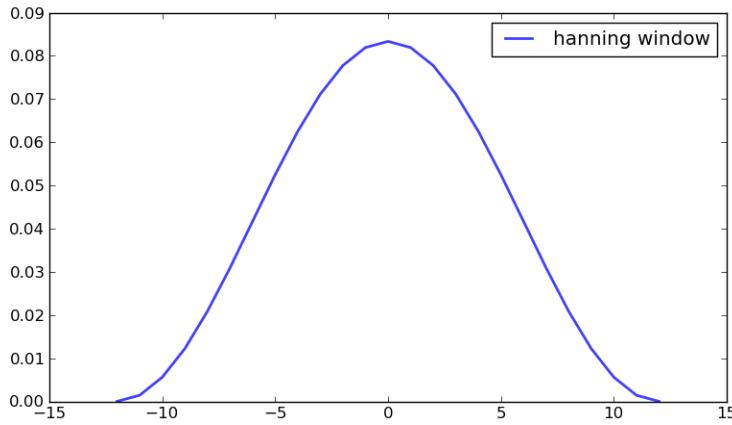
$$I_S(\omega_j) := \sum_{\ell=-p}^p w(\ell) I(\omega_{j+\ell}) \quad (3.101)$$

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 (3.101)



**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.jl` and are available via [QuantEcon](#)

The file `estspec.jl` are printed below

```
#=
Functions for working with periodograms of scalar data.

@author : Spencer Lyon <spencer.lyon@nyu.edu>

@date : 2014-08-21

References
-----

http://quant-econ.net/jl/estspec.html

=#
import DSP

"""
Smooth the data in x using convolution with a window of requested size and type.

##### Arguments

- `x::Array`: An array containing the data to smooth
- `window_len::Int(7)`: An odd integer giving the length of the window
- `window::String("hanning")`: A string giving the window type. Possible values are `flat`, `hanning`, `hamming`, `bartlett`, or `blackman`

##### Returns

- `out::Array`: The array of smoothed data
"""
function smooth(x::Array, window_len::Int=7, window::String="hanning")
    if length(x) < window_len
```

```

        throw(ArgumentError("Input vector length must be >= window length"))
    end

    if window_len < 3
        throw(ArgumentError("Window length must be at least 3."))
    end

    if iseven(window_len)
        window_len += 1
        println("Window length must be odd, reset to $window_len")
    end

    windows = @compat Dict("hanning" => DSP.hanning,
                           "hamming" => DSP.hamming,
                           "bartlett" => DSP.bartlett,
                           "blackman" => DSP.blackman,
                           "flat" => DSP.rect # moving average
                           )

    # Reflect x around x[0] and x[-1] prior to convolution
    k = round(Int, window_len / 2)
    xb = x[1:k] # First k elements
    xt = x[end-k+1:end] # Last k elements
    s = [reverse(xb); x; reverse(xt)]

    # === Select window values === #
    if !haskey(windows, window)
        msg = "Unrecognized window type '$window'"
        print(msg * " Defaulting to hanning")
        window = "hanning"
    end

    w = windows[window](window_len)

    return conv(w ./ sum(w), s)[window_len+1:end-window_len]
end

"Version of `smooth` where `window_len` and `window` are keyword arguments"
function smooth(x::Array; window_len::Int=7, window::String="hanning")
    smooth(x, window_len, window)
end

function periodogram(x::Vector)
    n = length(x)
    I_w = abs(fft(x)).^2 ./ n
    w = 2pi * (0:n-1) ./ n # Fourier frequencies

    # int rounds to nearest integer. We want to round up or take 1/2 + 1 to
    # make sure we get the whole interval from [0, pi]
    ind = iseven(n) ? round(Int, n / 2 + 1) : ceil(Int, n / 2)
    w, I_w = w[1:ind], I_w[1:ind]
    return w, I_w
end

```

```

function periodogram(x::Vector, window::String, window_len::Int=7)
    w, I_w = periodogram(x)
    I_w = smooth(I_w, window_len=window_len, window=window)
    return w, I_w
end

"""
Computes the periodogram

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.

##### Arguments

- `x::Array`: An array containing the data to smooth
- `window_len::Int(7)`: An odd integer giving the length of the window
- `window::String("hanning")`: A string giving the window type. Possible values
are `flat`, `hanning`, `hamming`, `bartlett`, or `blackman`


##### Returns

- `w::Array{Float64}`: Fourier frequencies at which the periodogram is evaluated
- `I_w::Array{Float64}`: The periodogram at frequencies `w`


"""

periodogram

"""

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.

##### Arguments

- `x::Array`: An array containing the data to smooth
- `window_len::Int(7)`: An odd integer giving the length of the window
- `window::String("hanning")`: A string giving the window type. Possible values
are `flat`, `hanning`, `hamming`, `bartlett`, or `blackman`


##### Returns

- `w::Array{Float64}`: Fourier frequencies at which the periodogram is evaluated
- `I_w::Array{Float64}`: The periodogram at frequencies `w`


"""

function ar_periodogram(x::Array, window::String="hanning", window_len::Int=7)
    # run regression
    x_current, x_lagged = x[2:end], x[1:end-1]  # x_t and x_{t-1}

```

```

coefs = linreg(x_lagged, x_current)

# get estimated values and compute residual
est = [ones(x_lagged) x_lagged] * coefs
e_hat = x_current - est

phi = coefs[2]

# compute periodogram on residuals
w, I_w = periodogram(e_hat, window, window_len)

# recolor and return
I_w = I_w ./ abs(1 - phi .* exp(im.*w)).^2

return w, I_w
end

```

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 `conv`

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 (3.100) — 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

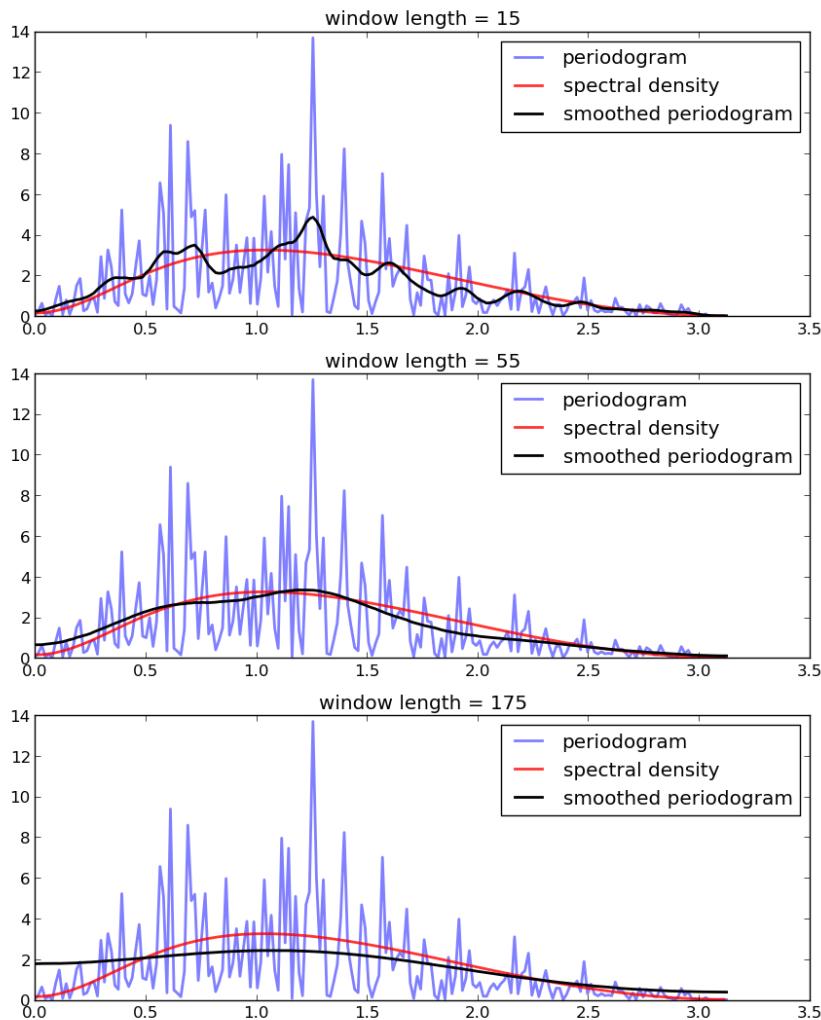
**Pre-Filtering and Smoothing** In the code listing *above* we showed three functions from the file `estspec.jl`

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 (3.101)

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

$$X_{t+1} = \mu + \phi X_t + \epsilon_{t+1} \quad (3.102)$$

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

$$f(\omega) = \left| \frac{1}{1 - \phi e^{i\omega}} \right|^2 g(\omega) \quad (3.103)$$

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.jl` — 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 (3.102) 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 (3.100) and there are 400 observations

For the smoothed periodogram, the window type is “hamming”

**Exercise 2** Replicate [this figure](#) (modulo randomness)

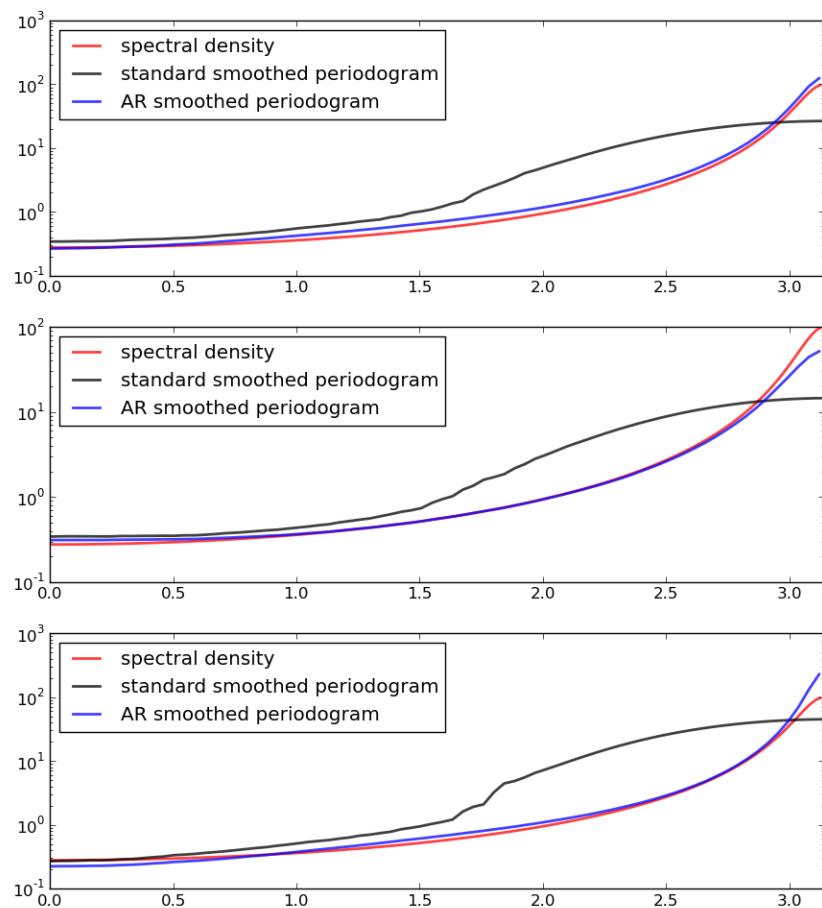
The model is as in equation (3.102), with  $\mu = 0$ ,  $\phi = -0.9$  and 150 observations in each time series

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



## 3.10 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

$$-\mathbb{E} \frac{1}{2} \sum_{t=0}^{\infty} \beta^t [(c_t - b_t)^2 + \ell_t^2] \quad (3.104)$$

subject to the budget constraint

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t p_t^0 [d_t + (1 - \tau_t) \ell_t + s_t - c_t] = 0 \quad (3.105)$$

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

$$c_t + g_t = d_t + \ell_t \quad (3.106)$$

A labor-consumption process  $\{\ell_t, c_t\}$  is called *feasible* if (3.106) holds for all  $t$

**Government budget constraint** Where  $p_t^0$  is a scaled Arrow-Debreu price, the time zero government budget constraint is

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t p_t^0 (s_t + g_t - \tau_t \ell_t) = 0 \quad (3.107)$$

**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 (3.107) 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 (3.107) in terms of exogenous variables and allocations
3. Maximize the household's objective function (3.104) subject to the constraint constructed in step 2 and the feasibility constraint (3.106)

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 (3.105), 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

$$p_t = \frac{b_t - c_t}{b_0 - c_0} \quad \text{and} \quad \tau_t = 1 - \frac{\ell_t}{b_t - c_t} \quad (3.108)$$

Substituting (3.108) into the government's budget constraint (3.107) yields

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t [(b_t - c_t)(s_t + g_t - \ell_t) + \ell_t^2] = 0 \quad (3.109)$$

The Ramsey problem now amounts to maximizing (3.104) subject to (3.109) and (3.106)

The associated Lagrangian is

$$\mathcal{L} = \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left\{ -\frac{1}{2} [(c_t - b_t)^2 + \ell_t^2] + \lambda [(b_t - c_t)(\ell_t - s_t - g_t) - \ell_t^2] + \mu_t [d_t + \ell_t - c_t - g_t] \right\} \quad (3.110)$$

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 (3.106) and working through the algebra, one can show that

$$\ell_t = \bar{\ell}_t - \nu m_t \quad \text{and} \quad c_t = \bar{c}_t - \nu m_t \quad (3.111)$$

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 (3.109) is  $(b_t - c_t)(s_t + g_t) - (b_t - c_t)\ell_t + \ell_t^2$

Using (3.111), 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 (3.109), we get

$$\mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t (b_t - \bar{c}_t)(g_t + s_t) \right\} + (\nu^2 - \nu) \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t 2m_t^2 \right\} = 0 \quad (3.112)$$

Although it might not be clear yet, we are nearly there:

- The two expectations terms in (3.112) 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 (3.111)
- Once we have the allocations, prices and the tax system can be derived from (3.108)

**Solving the Quadratic Term** Let's consider how to obtain the term  $\nu$  in (3.112)

If we can solve the two expected geometric sums

$$b_0 := \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t (b_t - \bar{c}_t)(g_t + s_t) \right\} \quad \text{and} \quad a_0 := \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t 2m_t^2 \right\} \quad (3.113)$$

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 (3.113)

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 (3.113), 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

$$q(x_0) = \mathbb{E} \sum_{t=0}^{\infty} \beta^t x_t' H x_t \quad (3.114)$$

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

$$q(x_0) = \sum_{t=0}^{\infty} \beta^t (P^t h)[j] \quad (3.115)$$

Here

- $P^t$  is the  $t$ -th power of the transition matrix  $P$
- $h$  is, with some abuse of notation, the vector  $(h(x^1), \dots, h(x^N))$
- $(P^t h)[j]$  indicates the  $j$ -th element of  $P^t h$

It can be shown that (3.115) 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

$$B_t := \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j p_{t+j}^t (\tau_{t+j} \ell_{t+j} - g_{t+j}) \quad (3.116)$$

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 `examples/lqramsey.jl` from the [main repository](#)

Description and clarifications are given below

```
#=
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.

@author : Spencer Lyon <spencer.lyon@nyu.edu>
```

```

@date: 2014-08-21

References
-----
Simple port of the file examples/lqramsey.py

http://quant-econ.net/lqramsey.html

=#
using QuantEcon
using PyPlot

abstract AbstractStochProcess

type ContStochProcess <: AbstractStochProcess
    A::Matrix
    C::Matrix
end

type DiscreteStochProcess <: AbstractStochProcess
    P::Matrix
    x_vals::Array
end

type Economy{SP <: AbstractStochProcess}
    bet::Real
    Sg::Matrix
    Sd::Matrix
    Sb::Matrix
    Ss::Matrix
    is_discrete::Bool
    proc::SP
end

type Path
    g
    d
    b
    s
    c
    l
    p
    tau
    rvn
    B
    R
    pi
    Pi

```

```

xi
end

function compute_exog_sequences(econ::Economy, x)
    # Compute exogenous variable sequences
    Sg, Sd, Sb, Ss = econ.Sg, econ.Sd, econ.Sb, econ.Ss
    g, d, b, s = [squeeze(S * x, 1) 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

    return g, d, b, s, Sm
end

function compute_allocation(econ::Economy, Sm, nu, x, b)
    Sg, Sd, Sb, Ss = econ.Sg, econ.Sd, econ.Sb, econ.Ss

    # 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 = squeeze(Sc * x, 1)
    l = squeeze(Sl * x, 1)
    p = squeeze((Sb - Sc) * x, 1) # Price without normalization
    tau = 1 .- l ./ (b .- c)
    rvn = l .* tau

    return Sc, Sl, c, l, p, tau, rvn
end

function compute_nu(a0, b0)
    disc = a0^2 - 4a0*b0

    if disc >= 0
        nu = 0.5 *(a0 - sqrt(disc)) / a0
    else
        println("There is no Ramsey equilibrium for these parameters.")
        error("Government spending (economy.g) too low")
    end

    # Test that the Lagrange multiplier has the right sign
    if nu * (0.5 - nu) < 0
        print("Negative multiplier on the government budget constraint.")
        error("Government spending (economy.g) too low")
    end

    return nu
end

```

```

function compute_Pi(B, R, rvn, g, xi)
    pi = B[2:end] - R[1:end-1] .* B[1:end-1] - rvn[1:end-1] + g[1:end-1]
    Pi = cumsum(pi .* xi)
    return pi, Pi
end

function compute_paths(econ::Economy{DiscreteStochProcess}, T)
    # simplify notation
    bet, Sg, Sd, Sb, Ss = econ.bet, econ.Sg, econ.Sd, econ.Sb, econ.Ss
    P, x_vals = econ.proc.P, econ.proc.x_vals

    state = mc_sample_path(P, 1, T)
    x = x_vals[:, state]

    # Compute exogenous sequence
    g, d, b, s, Sm = compute_exog_sequences(econ, x)

    # compute a0, b0
    ns = size(P, 1)
    F = eye(ns) - bet.*P
    a0 = (F \ ((Sm * x_vals)' .^ 2))[1] ./ 2
    H = ((Sb - Sd + Sg) * x_vals) .* ((Sg - Ss)*x_vals)
    b0 = (F \ H')[1] ./ 2

    # compute lagrange multiplier
    nu = compute_nu(a0, b0)

    # Solve for the allocation given nu and x
    Sc, Sl, c, l, p, tau, rvn = compute_allocation(econ, Sm, nu, x, b)

    # compute remaining variables
    H = ((Sb - Sc)*x_vals) .* ((Sl - Sg)*x_vals) - (Sl*x_vals).^2
    temp = squeeze(F*H', 2)
    B = temp[state] ./ p
    H = squeeze(P[state, :] * ((Sb - Sc)*x_vals)', 2)
    R = p ./ (bet .* H)
    temp = squeeze(P[state, :] * ((Sb - Sc) * x_vals)', 2)
    xi = p[2:end] ./ temp[1:end-1]

    # compute pi
    pi, Pi = compute_Pi(B, R, rvn, g, xi)

    Path(g, d, b, s, c, l, p, tau, rvn, B, R, pi, Pi, xi)
end

function compute_paths(econ::Economy{ContStochProcess}, T)
    # simplify notation
    bet, Sg, Sd, Sb, Ss = econ.bet, econ.Sg, econ.Sd, econ.Sb, econ.Ss
    A, C = econ.proc.A, econ.proc.C

```

```

# Generate an initial condition x0 satisfying x0 = A x0
nx, nx = size(A)
x0 = null((eye(nx) - A))
x0 = x0[end] < 0 ? -x0 : x0
x0 = x0 ./ x0[end]
x0 = squeeze(x0, 2)

# Generate a time series x of length T starting from x0
nx, nw = size(C)
x = zeros(nx, T)
w = randn(nw, T)
x[:, 1] = x0
for t=2:T
    x[:, t] = A *x[:, t-1] + C * w[:, t]
end

# Compute exogenous sequence
g, d, b, s, Sm = compute_exog_sequences(econ, x)

# compute a0 and b0
H = Sm'Sm
a0 = 0.5 * var_quadratic_sum(A, C, H, bet, x0)
H = (Sb - Sd + Sg)'*(Sg + Ss)
b0 = 0.5 * var_quadratic_sum(A, C, H, bet, x0)

# compute lagrange multiplier
nu = compute_nu(a0, b0)

# Solve for the allocation given nu and x
Sc, Sl, c, l, p, tau, rvn = compute_allocation(econ, Sm, nu, x, b)

# compute remaining variables
H = Sl'Sl - (Sb - Sc)'*(Sl - Sg)
L = Array(Float64, T)
for t=1:T
    L[t] = var_quadratic_sum(A, C, H, bet, x[:, t])
end
B = L ./ p
Rinv = squeeze(bet .* (Sb - Sc)*A*x, 1) ./ p
R = 1 ./ Rinv
AF1 = (Sb - Sc) * x[:, 2:end]
AF2 = (Sb - Sc) * A * x[:, 1:end-1]
xi = AF1 ./ AF2
xi = squeeze(xi, 1)

# compute pi
pi, Pi = compute_Pi(B, R, rvn, g, xi)

Path(g, d, b, s, c, l, p, tau, rvn, B, R, pi, Pi, xi)
end

function gen_fig_1(path::Path)

```

```

T = length(path.c)

num_rows, num_cols = 2, 2
fig, axes = subplots(num_rows, num_cols, figsize=(14, 10))
plt.subplots_adjust(hspace=0.4)
for i=1:num_rows
    for j=1:num_cols
        axes[i, j][:grid]()
        axes[i, j][:set_xlabel]("Time")
    end
end

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, gout expenditure and revenue
ax = axes[1, 1]
ax[:plot](path.rvn, label=L"\tau_t \ell_t"; p_args...)
ax[:plot](path.g, label=L"g_t"; p_args...)
ax[:plot](path.c, label=L"c_t"; p_args...)
ax[:legend](ncol=3; legend_args...)

# Plot gout expenditure and debt
ax = axes[1, 2]
ax[:plot](1:T, path.rvn, label=L"\tau_t \ell_t"; p_args...)
ax[:plot](1:T, path.g, label=L"g_t"; p_args...)
ax[:plot](1:T-1, path.B[2:end], label=L"B_{t+1}"; p_args...)
ax[:legend](ncol=3; legend_args...)

# Plot risk free return
ax = axes[2, 1]
ax[:plot](1:T, path.R - 1, label=L"R_{t - 1}"; p_args...)
ax[:legend](ncol=1; legend_args...)

# Plot revenue, expenditure and risk free rate
ax = axes[2, 2]
ax[:plot](1:T, path.rvn, label=L"\tau_t \ell_t"; p_args...)
ax[:plot](1:T, path.g, label=L"g_t"; p_args...)
ax[:plot](1:T-1, path.pi, label=L"\pi_{t+1}"; p_args...)
ax[:legend](ncol=3; legend_args...)

plt.show()

end

function gen_fig_2(path::Path)
    T = length(path.c)

    # Prepare axes
    num_rows, num_cols = 2, 1

```

```

fig, axes = subplots(num_rows, num_cols, figsize=(10, 10))
plt.subplots_adjust(hspace=0.5)
bbox = (0., 1.02, 1., .102)
legend_args = { :bbox_to_anchor => bbox, :loc => 3, :mode => :expand}
p_args = { :lw => 2, :alpha => 0.7}

# Plot adjustment factor
ax = axes[1]
ax[:plot](2:T, path.xi, label=L"\xi_t"; p_args...)
ax[:grid]()
ax[:set_xlabel]("Time")
ax[:legend](ncol=1; legend_args...)

# Plot adjusted cumulative return
ax = axes[2]
ax[:plot](2:T, path.Pi, label=L"\Pi_t"; p_args...)
ax[:grid]()
ax[:set_xlabel]("Time")
ax[:legend](ncol=1; legend_args...)

plt.show()
end

```

**Comments on the Code** The function `var_quadratic_sum` from `QuantEcon.jl` is for computing the value of (3.114) when the exogenous process  $\{x_t\}$  is of the VAR type described *above*

This code defines two Types: `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

### 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 `examples/lqramsey_ar1.jl`

```
#=
Example 1: Govt spending is AR(1) and state is (g, 1).
@author : Spencer Lyon <spencer.lyon@nyu.edu>
@date: 2014-08-21
References
-----
Simple port of the file examples/lqramsey_ar1.py
http://quant-econ.net/lqramsey.html

=#
include("lqramsey.jl")

# == Parameters ==
bet = 1 / 1.05
rho, mg = .7, .35
A = eye(2)
A = [rho mg*(1 - rho); 0.0 1.0]
C = [sqrt(1 - rho^2)*mg/10 0.0]'
Sg = [1.0 0.0]
Sd = [0.0 0.0]
Sb = [0 2.135]
Ss = [0.0 0.0]
discrete = false
proc = ContStochProcess(A, C)

econ = Economy(bet, Sg, Sd, Sb, Ss, discrete, proc)
T = 50

path = compute_paths(econ, T)

gen_fig_1(path)
```

Running the program produces the figure

The legends on the figures indicate the variables being tracked

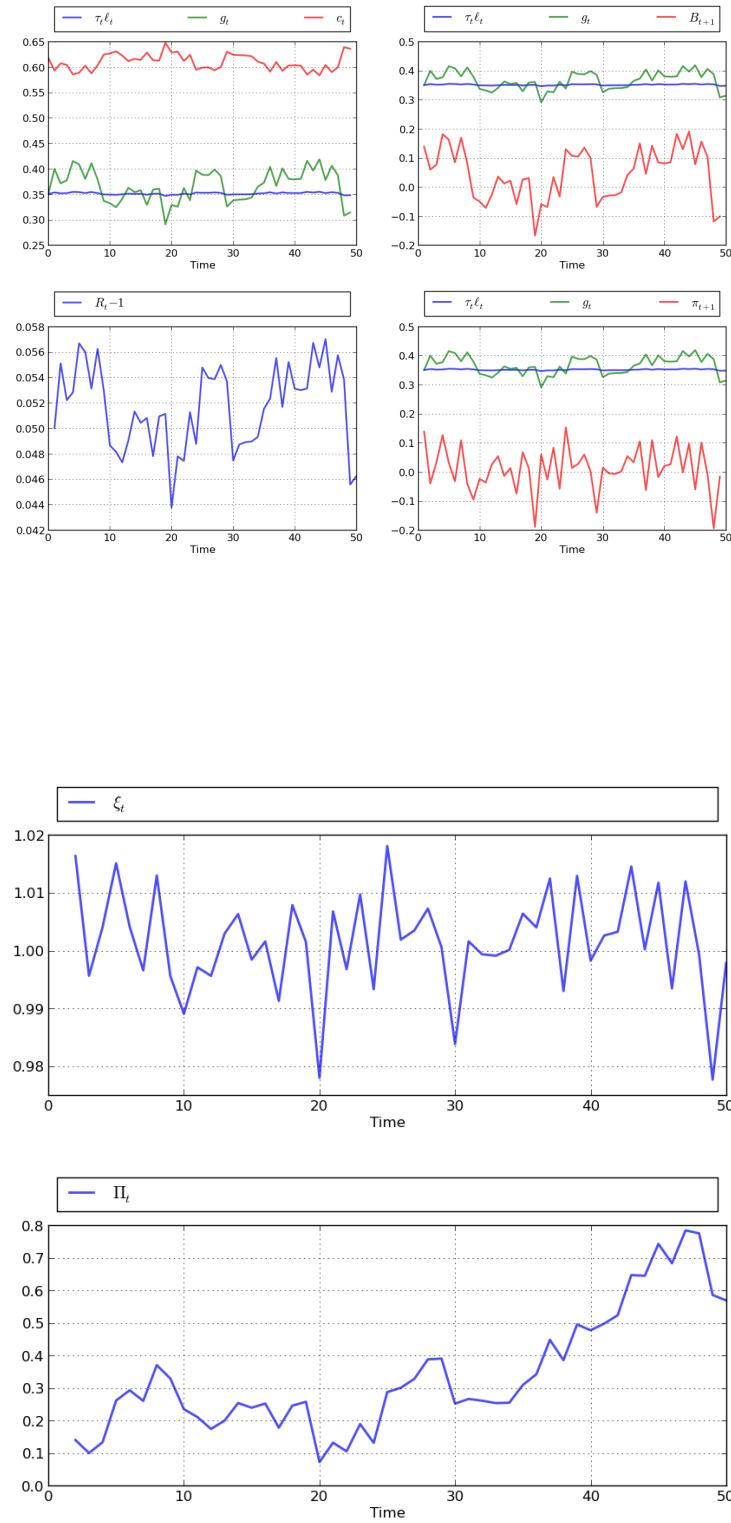
Most obvious from the figure is tax smoothing in the sense that tax revenue is much less variable than government expenditure

After running the code above, if you then execute `gen_fig_2(path)` from your Julia console 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 `examples/lqramsey_discrete.jl`



```

#=

Example 2: LQ Ramsey model with discrete exogenous process.

@author : Spencer Lyon <spencer.lyon@nyu.edu>

@date: 2014-08-21

References
-----

Simple port of the file examples/lqramsey_discrete.py

http://quant-econ.net/lqramsey.html

=#
include("lqramsey.jl")

# Parameters
bet = 1 / 1.05
P = [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 = [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 = [1.0 0.0 0.0 0.0 0.0]
Sd = [0.0 1.0 0.0 0.0 0.0]
Sb = [0.0 0.0 1.0 0.0 0.0]
Ss = [0.0 0.0 0.0 1.0 0.0]
discrete = true
proc = DiscreteStochProcess(P, x_vals)

econ = Economy(bet, Sg, Sd, Sb, Ss, discrete, proc)
T = 15

path = compute_paths(econ, T)

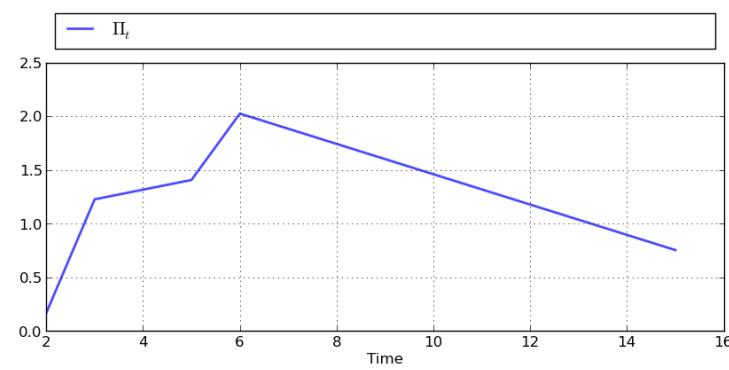
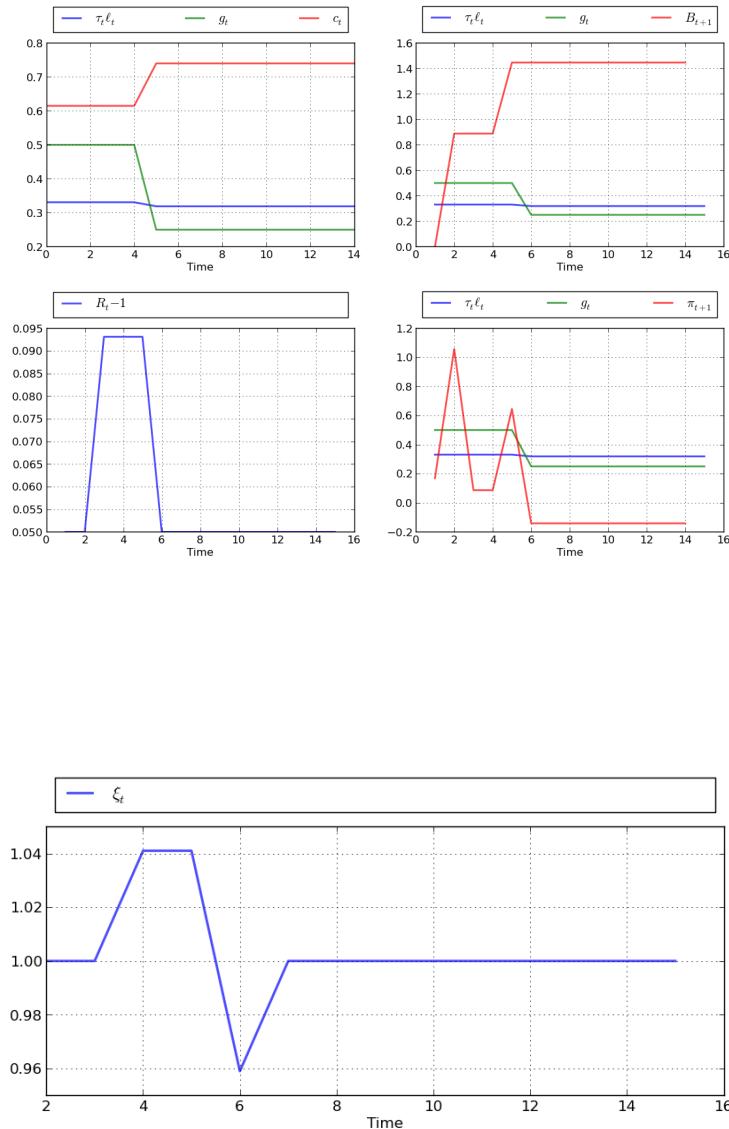
gen_fig_1(path)

```

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](#)

## 3.11 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

**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]

---

<sup>1</sup> We could also call a competitive equilibrium a rational expectations equilibrium.

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

$$p_t = A_0 - A_1 Q_t, \quad A_0 > 0, A_1 > 0 \quad (3.117)$$

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

$$\sum_{t=0}^{\infty} \beta^t \left\{ p_t q_t - \frac{d}{2} (q_{t+1} - q_t)^2 - \tau_t q_t \right\} \quad (3.118)$$

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

$$u_t = \frac{\beta}{d} p_{t+1} + \beta u_{t+1} - \frac{\beta}{d} \tau_{t+1}, \quad t = 0, 1, \dots \quad (3.119)$$

To compute a competitive equilibrium, it is appropriate to take (3.119), eliminate  $p_t$  in favor of  $Q_t$  by using (3.117), and then set  $q_t = Q_t$

This last step makes the representative firm be representative<sup>2</sup>

We arrive at

$$u_t = \frac{\beta}{d}(A_0 - A_1 Q_{t+1}) + \beta u_{t+1} - \frac{\beta}{d} \tau_{t+1} \quad (3.120)$$

$$Q_{t+1} = Q_t + u_t \quad (3.121)$$

**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 (3.117), (3.120), and (3.121)

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 (3.120) entirely through its impact on  $u_1$

A continuation competitive equilibrium can be indexed by a  $u_1$  that satisfies (3.120)

In the spirit of [KP80], 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

### Ramsey Problem

The planner's objective is cast in terms of consumer surplus net of the firm's adjustment costs

---

<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  $du_t$  of  $u_t = q_{t+1} - q_t$ . This choice would align better with how Chang [Cha98] chose to express his competitive equilibrium recursively. But given  $(u_t, Q_t)$ , the representative firm knows  $(Q_{t+1}, \tau_{t+1})$ , 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  $u_t$ .

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

$$A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 \quad (3.122)$$

At time  $t = 0$ , a Ramsey planner faces the intertemporal budget constraint

$$\sum_{t=1}^{\infty} \beta^t \tau_t Q_t = G_0 \quad (3.123)$$

Note that (3.123) 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

$$\sum_{t=0}^{\infty} \beta^t \left\{ A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 \right\} \quad (3.124)$$

subject to (3.123)

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 (3.124) subject to (3.120), (3.121), and (3.123)

To formulate this problem as a Lagrangian, attach a Lagrange multiplier  $\mu$  to the budget constraint (3.123)

Then the planner chooses  $\{u_t\}_{t=0}^{\infty}, \{\tau_t\}_{t=1}^{\infty}$  to maximize and the Lagrange multiplier  $\mu$  to minimize

$$\sum_{t=0}^{\infty} \beta^t (A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2) + \mu \left[ \sum_{t=0}^{\infty} \beta^t \tau_t Q_t - G_0 - \tau_0 Q_0 \right] \quad (3.125)$$

subject to and (3.120) and (3.121)

The Ramsey problem is a special case of the linear quadratic dynamic Stackelberg problem analyzed in this lecture

The key implementability conditions are (3.120) 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

$$\max_{\{y_t, \tau_{t+1}\}} - \sum_{t=0}^{\infty} \beta^t y_t' R y_t \tag{3.126}$$

subject to  $z_0$  given and the law of motion

$$y_{t+1} = A y_t + B \tau_{t+1} \tag{3.127}$$

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

$$v(y_t) = \max_{\tau_{t+1}} \{-y_t' R y_t + \beta v(y_{t+1})\} \quad (3.128)$$

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 (3.128) has solution

$$v(y_t) = -y_t' P y_t$$

where

- $P$  solves the algebraic matrix Riccati equation  $P = R + \beta A' P A - \beta A' P B (B' P B)^{-1} B' P A$
- the optimal policy function is given by  $\tau_{t+1} = -F y_t$  for  $F = (B' P B)^{-1} B' P A$

Now we turn to subproblem 1.

Evidently the optimal choice of  $u_0$  satisfies  $\frac{\partial v}{\partial u_0} = 0$

If we partition  $P$  as

$$P = \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix}$$

then we have

$$0 = \frac{\partial}{\partial u_0} (z'_0 P_{11} z_0 + z'_0 P_{12} u_0 + u'_0 P_{21} z_0 + u'_0 P_{22} u_0) = P'_{12} z_0 + P_{21} u_0 + 2P_{22} u_0$$

which implies

$$u_0 = -P_{22}^{-1} P_{21} z_0 \quad (3.129)$$

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

$$u_0 = v(Q_0 | \mu) \quad (3.130)$$

that obeys (3.129) and the following updating equations for  $t \geq 0$ :

$$\tau_{t+1} = \tau(Q_t, u_t | \mu) \quad (3.131)$$

$$Q_{t+1} = Q_t + u_t \quad (3.132)$$

$$u_{t+1} = u(Q_t, u_t | \mu) \quad (3.133)$$

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

```
#=
```

*In the following, ``what`` and ``tauhat`` are what the planner would choose if he could reset at time  $t$ , ``uhatdif`` and ``tauhatdif`` are the difference between those and what the planner is constrained to choose. The variable ``mu`` is the Lagrange multiplier associated with*

```

the constraint at time t.

For more complete description of inputs and outputs see the website.

@author : Spencer Lyon <spencer.lyon@nyu.edu>

@date: 2014-08-21

References
-----
Simple port of the file examples/evans_sargent.py

http://quant-econ.net/hist_dep_policies.html

=#
using QuantEcon
using Optim
using PyPlot

type HistDepRamsey
    # These are the parameters of the economy
    A0::Real
    A1::Real
    d::Real
    Q0::Real
    tau0::Real
    mu0::Real
    bet::Real

    # These are the LQ fields and stationary values
    R::Matrix
    A::Matrix
    B::Matrix
    Q::Real
    P::Matrix
    F::Matrix
    lq::LQ
end

type RamseyPath
    y::Matrix
    uhat::Vector
    uhatdif::Vector
    tauhat::Vector
    tauhatdif::Vector
    mu::Vector
    G::Vector
    GPay::Vector
end

```

```

function HistDepRamsey(A0, A1, d, Q0, tau0, mu, bet)
    # Create Matrices for solving Ramsey problem
    R = [0.0 -A0/2 0.0 0.0
          -A0/2 A1/2 -mu/2 0.0
          0.0 -mu/2 0.0 0.0
          0.0 0.0 0.0 d/2]

    A = [1.0 0.0 0.0 0.0
          0.0 1.0 0.0 1.0
          0.0 0.0 0.0 0.0
          -A0/d A1/d 0.0 A1/d+1.0/bet]

    B = [0.0 0.0 1.0 1.0/d]'

    Q = 0.0

    # Use LQ to solve the Ramsey Problem.
    lq = LQ(Q, -R, A, B, bet=bet)

    P, F, _d = stationary_values(lq)

    HistDepRamsey(A0, A1, d, Q0, mu0, bet, R, A, B, Q, P, F, lq)
end

function compute_G(hdr::HistDepRamsey, mu)
    # simplify notation
    Q0, tau0, P, F, d, A, B = hdr.Q0, hdr.tau0, hdr.P, hdr.F, hdr.d, hdr.A, hdr.B
    bet = hdr.bet

    # Need y_0 to compute government tax revenue.
    u0 = compute_u0(hdr, P)
    y0 = vcat([1.0 Q0 tau0]', u0)

    # Define A_F and S matricies
    AF = A - B * F
    S = [0.0 1.0 0.0 0]' * [0.0 0.0 1.0 0]

    # Solves equation (25)
    Omega = solve_discrete_lyapunov(sqrt(bet) .* AF', bet .* AF' * S * AF)
    T0 = y0' * Omega * y0

    return T0[1], A, B, F, P
end

function compute_u0(hdr::HistDepRamsey, P::Matrix)
    # simplify notation
    Q0, tau0 = hdr.Q0, hdr.tau0

    P21 = P[4, 1:3]
    P22 = P[4, 4]
    z0 = [1.0 Q0 tau0]'

```

```

u0 = -P22^(-1) .* P21*(z0)

    return u0[1]
end

function init_path(hdr::HistDepRamsey, mu0, T::Int=20)
    # Construct starting values for the path of the Ramsey economy
    G0, A, B, F, P = compute_G(hdr, mu0)

    # Compute the optimal u0
    u0 = compute_u0(hdr, P)

    # Initialize vectors
    y = Array(Float64, 4, T)
    uhat      = Array(Float64, T)
    uhatdif   = Array(Float64, T)
    tauhat    = Array(Float64, T)
    tauhatdif = Array(Float64, T-1)
    mu        = Array(Float64, T)
    G         = Array(Float64, T)
    GPay      = Array(Float64, T)

    # Initial conditions
    G[1] = G0
    mu[1] = mu0
    uhatdif[1] = 0.0
    uhat[1] = u0
    y[:, 1] = vcat([1.0 hdr.Q0 hdr.tau0]', u0)

    return RamseyPath(y, uhat, uhatdif, tauhat, tauhatdif, mu, G, GPay)
end

function compute_ramsey_path!(hdr::HistDepRamsey, rp::RamseyPath)
    # simplify notation
    y, uhat, uhatdif, tauhat, = rp.y, rp.uhat, rp.uhatdif, rp.tauhat
    tauhatdif, mu, G, GPay = rp.tauhatdif, rp.mu, rp.G, rp.GPay
    bet = hdr.bet

    G0, A, B, F, P = compute_G(hdr, mu[1])

    for t=2:T
        # iterate government policy
        y[:, t] = (A - B * F) * y[:, t-1]

        # update G
        G[t] = (G[t-1] - bet*y[2, t]*y[3, t])/bet
        GPay[t] = bet.*y[2, t]*y[3, t]

        #=
        Compute the mu if the government were able to reset its plan
    end
end

```

```

ff is the tax revenues the government would receive if they reset the
plan with Lagrange multiplier mu minus current G
=#
ff(mu) = abs(compute_G(hdr, mu)[1]-G[t])

# find ff = 0
mu[t] = optimize(ff, mu[t-1]-1e4, mu[t-1]+1e4).minimum
temp, Atemp, Btemp, Ftemp, Ptemp = compute_G(hdr, mu[t])

# Compute alternative decisions
P21temp = Ptemp[4, 1:3]
P22temp = P[4, 4]
uhat[t] = (-P22temp^(-1) .* P21temp * y[1:3, t])[1]

yhat = (Atemp-Btemp * Ftemp) * [y[1:3, t-1], uhat[t-1]]
tauhat[t] = yhat[3]
tauhatdif[t-1] = tauhat[t] - y[3, t]
uhatdif[t] = uhat[t] - y[3, t]
end

return rp
end

function plot1(rp::RamseyPath)
    tt = 1:length(rp.mu) # tt is used to make the plot time index correct.
    y = rp.y

    n_rows = 3
    fig, axes = subplots(n_rows, 1, figsize=(10, 12))

    subplots_adjust(hspace=0.5)
    for ax in axes
        ax[:grid]()
        ax[:set_xlim](0, 15)
    end

    bbox = (0., 1.02, 1., .102)
    legend_args = {:bbox_to_anchor => bbox, :loc => 3, :mode => "expand"}
    p_args = {:lw => 2, :alpha => 0.7}

    ax = axes[1]
    ax[:plot](tt, squeeze(y[2, :], 1), "b-", label="output"; p_args...)
    ax[:set_ylabel](L"\$Q\$", fontsize=16)
    ax[:legend](ncol=1; legend_args...)

    ax = axes[2]
    ax[:plot](tt, squeeze(y[3, :], 1), "b-", label="tax rate"; p_args...)
    ax[:set_ylabel](L"\$\tau\$", fontsize=16)
    ax[:set_yticks]((0.0, 0.2, 0.4, 0.6, 0.8))
    ax[:legend](ncol=1; legend_args...)

    ax = axes[3]

```

```

ax[:plot](tt, squeeze(y[4, :], 1), "b-", label="first difference in output";
          p_args...)
ax[:set_ylabel](L"$u$", fontsize=16)
ax[:set_yticks]((0, 100, 200, 300, 400))
ax[:legend](ncol=1; legend_args...)
ax[:set_xlabel](L"time", fontsize=16)

plt.show()
end

function plot2(rp::RamseyPath)
    y, uhatdif, tauhatdif, mu = rp.y, rp.uhatdif, rp.tauhatdif, rp.mu
    G, GPay = rp.G, rp.GPay
    T = length(rp.mu)
    tt = 1:T # tt is used to make the plot time index correct.
    tt2 = 1:T-1

    n_rows = 4
    fig, axes = subplots(n_rows, 1, figsize=(10, 16))

    plt.subplots_adjust(hspace=0.5)
    for ax in axes
        ax[:grid](alpha=.5)
        ax[:set_xlim](-0.5, 15)
    end

    bbox = (0., 1.02, 1., .102)
    legend_args = {:bbox_to_anchor => bbox, :loc => 3, :mode => "expand"}
    p_args = {:lw => 2, :alpha => 0.7}

    ax = axes[1]
    ax[:plot](tt2, tauhatdif,
              label="time inconsistency differential for tax rate"; p_args...)
    ax[:set_ylabel](L"\Delta\tau", fontsize=16)
    ax[:set_yticks]((0.0, 0.4, 0.8, 1.2))
    ax[:legend](ncol=1; legend_args...)

    ax = axes[2]
    ax[:plot](tt, uhatdif,
              label=L"time inconsistency differential for $u$"; p_args...)
    ax[:set_ylabel](L"\Delta u", fontsize=16)
    ax[:set_yticks]((-3.0, -2.0, -1.0, 0.0))
    ax[:legend](ncol=1; legend_args...)

    ax = axes[3]
    ax[:plot](tt, mu, label="Lagrange multiplier"; p_args...)
    ax[:set_ylabel](L"\mu", fontsize=16)
    ax[:set_yticks]((2.34e-3, 2.43e-3, 2.52e-3))
    ax[:legend](ncol=1; legend_args...)

    ax = axes[4]
    ax[:plot](tt, G, label="government revenue"; p_args...)
    ax[:set_ylabel](L"G", fontsize=16)

```

```

ax[:set_yticks]((9200, 9400, 9600, 9800))
ax[:legend](ncol=1; legend_args...)

ax[:set_xlabel](L"time", fontsize=16)

plt.show()
end

# Primitives
T      = 20
A0    = 100.0
A1    = 0.05
d     = 0.20
bet   = 0.95

# Initial conditions
mu0  = 0.0025
Q0   = 1000.0
tau0 = 0.0

# Solve Ramsey problem and compute path
hdr = HistDepRamsey(A0, A1, d, Q0, tau0, mu0, bet)
rp = init_path(hdr, mu0, T)
compute_ramsey_path!(hdr, rp) # updates rp in place
plot1(rp)
plot2(rp)

```

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 outcome 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>

$$\tau_{t+1} = -248.0624 - 0.1242Q_t - 0.3347u_t \quad (3.134)$$

Notice how the Ramsey plan calls for a high tax at  $t = 1$  followed by a perpetual stream of lower taxes

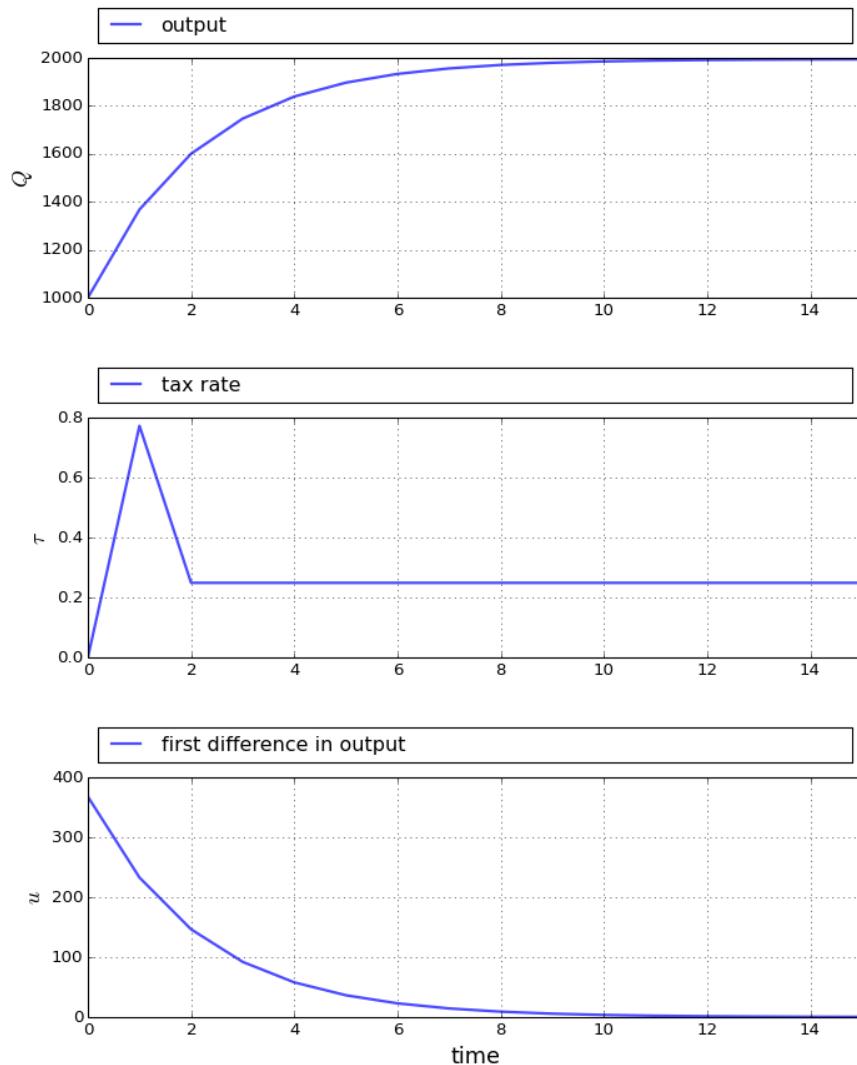
Taxing heavily at first, less later expresses time-inconsistency of the optimal plan for  $\{\tau_{t+1}\}_{t=0}^\infty$

We'll characterize this formally after first discussing how to compute  $\mu$ .

**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  $Q_t = e'_Q y_t$

---

<sup>4</sup> As promised,  $\tau_t$  does not appear in the Ramsey planner's decision rule for  $\tau_{t+1}$ .



Evidently  $Q_t \tau_t = y'_t e_Q e'_\tau y_t = y'_t S y_t$  where  $S := e_Q e'_\tau$

We want to compute

$$T_0 = \sum_{t=1}^{\infty} \beta^t \tau_t Q_t = \tau_1 Q_1 + \beta T_1$$

where  $T_1 = \sum_{t=2}^{\infty} \beta^{t-1} Q_t \tau_t$

The present values  $T_0$  and  $T_1$  are connected by

$$T_0 = \beta y'_0 A'_F S A_F y_0 + \beta T_1$$

Guess a solution that takes the form  $T_t = y'_t \Omega y_t$ , then find an  $\Omega$  that satisfies

$$\Omega = \beta A'_F S A_F + \beta A'_F \Omega A_F \quad (3.135)$$

Equation (3.135) 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  $\mu$ ; if  $T_0 < G_0$ , 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 (3.120), (3.121), and (3.123)

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

$$w(Q_0, u_0 | \mu_0) = \sum_{t=0}^{\infty} \beta^t \left\{ A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 \right\} \quad (3.136)$$

where

- $\{Q_t, u_t\}_{t=0}^{\infty}$  are evaluated under the Ramsey plan whose recursive representation is given by (3.131), (3.132), (3.133)
- $\mu_0$  is the value of the Lagrange multiplier that assures budget balance, computed as described [above](#)

Evidently, these continuation values satisfy the recursion

$$w(Q_t, u_t | \mu_0) = A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 + \beta w(Q_{t+1}, u_{t+1} | \mu_0) \quad (3.137)$$

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 (3.131), (3.132), (3.133)

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

$$G_t = \beta^{-t} (G_0 - \sum_{s=1}^t \beta^s \tau_s Q_s) \quad (3.138)$$

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 (3.138)

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

$$w(Q_t, v(Q_t | \check{\mu}) | \check{\mu}) > w(Q_t, u_t | \mu_0) \quad (3.139)$$

Inequality (3.139) expresses a continuation Ramsey planner’s incentive to deviate from a time 0 Ramsey plan by

1. resetting  $u_t$  according to (3.130)
2. adjusting the Lagrange multiplier on the continuation appropriately to account for tax revenues already collected <sup>6</sup>

Inequality (3.139) 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 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  $\tilde{G}_1$ .

**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

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 (3.129):

$$\check{u}_t = -P_{22}^{-1}(\check{\mu}_t) P_{21}(\check{\mu}_t) z_t$$

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 (3.129)

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

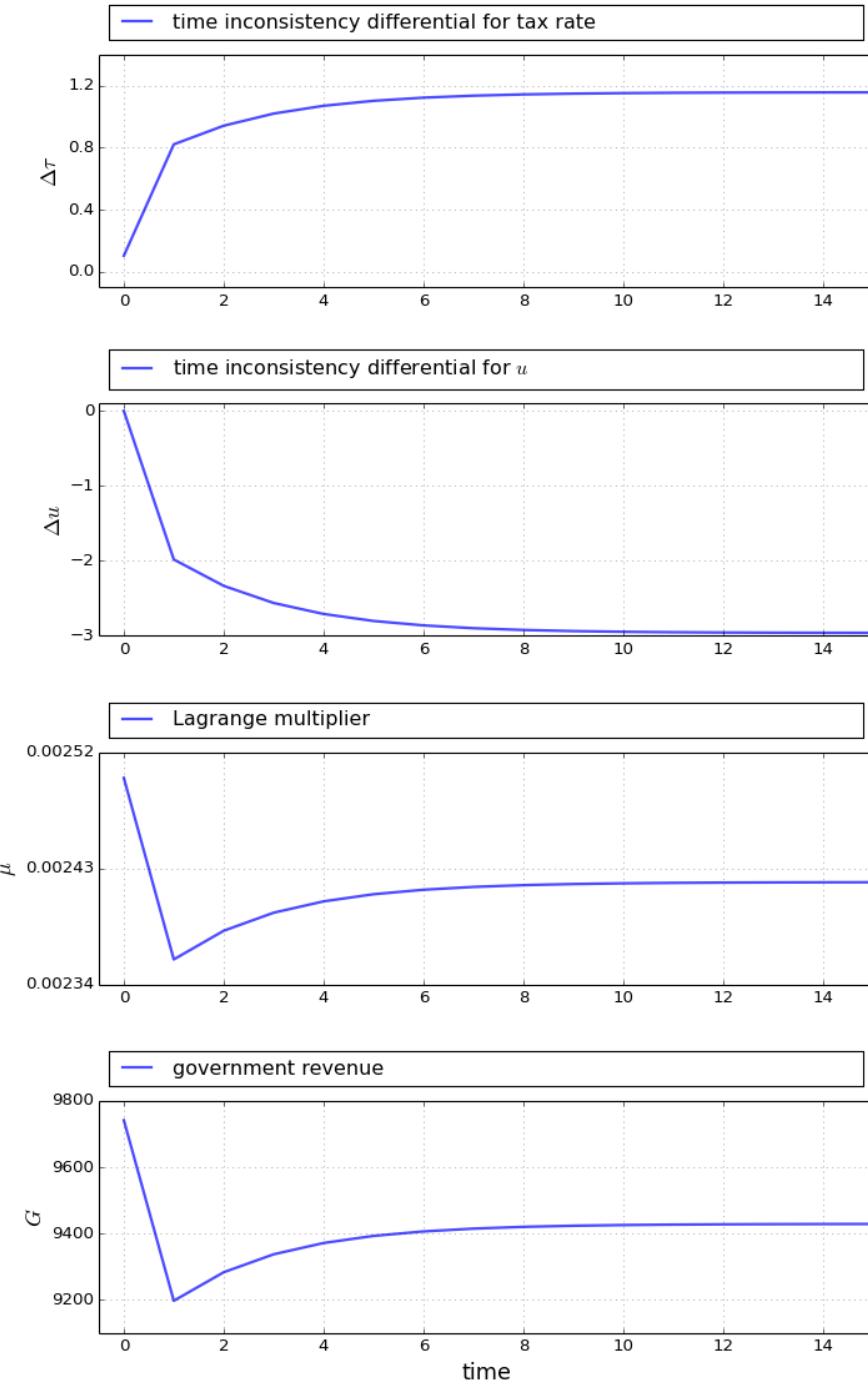
The intuition here is that the government is required to finance a given present value of expenditures with distorting taxes  $\tau$

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

---

<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 can also been seen in the third panel of *the second figure*, where the government pays off a significant portion of the debt using the first period tax rate

The similarities between the graphs in the last two panels of *the second figure* reveals that there is a one-to-one mapping between  $G$  and  $\mu$

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

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 (3.137), to define the continuation value function for the government:

$$J_t = A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 + \beta J_{t+1}(\tau_{t+1}, G_{t+1}) \quad (3.140)$$

This differs from (3.137) 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

$$G_t = \beta\tau_{t+1}Q_{t+1} + \beta G_{t+1}, \quad t \geq 0 \quad (3.141)$$

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 (3.141)

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 discounted present value promised to the Ramsey planner and take it as an initial condition.

Then after choosing  $u_0$  according to

$$u_0 = v(Q_0, G_0, J_0), \quad (3.142)$$

choose subsequent taxes, outputs, and continuation values according to recursions that can be represented as

$$\hat{\tau}_{t+1} = \tau(Q_t, u_t, G_t, J_t) \quad (3.143)$$

$$u_{t+1} = \xi(Q_t, u_t, G_t, J_t, \tau_{t+1}) \quad (3.144)$$

$$G_{t+1} = \beta^{-1}G_t - \tau_{t+1}Q_{t+1} \quad (3.145)$$

$$J_{t+1}(\tau_{t+1}, G_{t+1}) = \nu(Q_t, u_t, G_{t+1}, J_t, \tau_{t+1}) \quad (3.146)$$

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

<sup>8</sup> This choice is the key to what [LS12] call ‘dynamic programming squared’.

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

$$J_t = A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 + \beta J_{t+1}(\hat{\tau}_{t+1}, \hat{G}_{t+1}) \quad (3.147)$$

$$\geq A_0 Q_t - \frac{A_1}{2} Q_t^2 - \frac{d}{2} u_t^2 + \beta J_{t+1}(\tau_{t+1}, G_{t+1}) \quad (3.148)$$

for all tax rates  $\tau_{t+1} \in \mathbb{R}$  available to the government

Here  $\hat{G}_{t+1} = \frac{G_t - \hat{\tau}_{t+1} Q_{t+1}}{\beta}$

- Inequality expresses that continuation values adjust to deviations in ways that discourage the government from deviating from the prescribed  $\hat{\tau}_{t+1}$
- Inequality (3.147) indicates that *two* continuation values  $J_{t+1}$  contribute to sustaining time  $t$  promised value  $J_t$ 
  - $J_{t+1}(\hat{\tau}_{t+1}, \hat{G}_{t+1})$  is the continuation value when the government chooses to confirm the private sector's expectation, formed according to the decision rule (3.143)<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 (3.147) 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 (3.147) 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 (3.143), (3.144), (3.145)

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 (3.143), (3.144), (3.145) that attains that value

In many cases, there is **set** a 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

---

<sup>9</sup> Note the double role played by (3.143): as decision rule for the government and as the private sector's rule for forecasting government actions.

### Concluding remarks

The term ‘optimal policy’, which pervades an important applied monetary economics literature, means different things under different timing protocols

Under the ‘static’ Ramsey timing protocol (i.e., choose a sequence once-and-for-all), we obtain a unique plan

Here the phrase ‘optimal policy’ seems to fit well, since the Ramsey planner optimally reaps early benefits from influencing the private sector’s beliefs about the government’s later actions

When we adopt the sequential timing protocol associated with credible public policies, ‘optimal policy’ is a more ambiguous description

There is a multiplicity of credible plans

True, the theory explains how it is optimal for the government to confirm the private sector’s expectations about its actions along a credible plan

But some credible plans have very bad outcomes

These bad outcomes are central to the theory because it is the presence of bad credible plans that makes possible better ones by sustaining the low continuation values that appear in the second line of incentive constraint (3.147)

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

<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.

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

## 3.12 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

---

<sup>14</sup> See [Bas05] and [ACK10].

- countercyclical trade balance
- 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 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \quad (3.149)$$

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 (3.149)

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
- The cost of the contract is  $qB'$ 
  - if  $B' < 0$ , then  $-qB'$  units of the good are received in the current period, for a promise to repay  $-B$  units next period
  - the price  $q = q(B', y)$  will depend on both  $B'$  and  $y$  in equilibrium

Earnings on the government portfolio are paid lump sum to households

Conditional on absence of default, the resource constraint for the economy at any point in time is

$$c = y + B - q(B', y)B' \quad (3.150)$$

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 the government defaults with probability  $\delta$ , expected net payoffs are

$$qB' - \frac{1-\delta}{1+r}B' \quad (3.151)$$

**Government's decisions** At each point in time  $t$ , the government decides between

1. defaulting
2. meeting its current obligations and purchasing or selling an optimal quantity of its one-period sovereign debt

Defaulting means failing to repay its current obligations

If the government defaults in the current period, then consumption equals current output

A sovereign default has two further consequences:

1. Output falls from  $y$  to  $h(y)$ , where  $0 \leq h(y) \leq y$
2. The country loses access to foreign credit

While in a state of default, the economy regains access to foreign credit in each subsequent period with probability  $\theta$

### Equilibrium

Informally, a recursive equilibrium for this economy 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 satisfies the resource constraint
2. The government maximizes household utility taking into account
  - the resource constraint
  - the effect of its choices on the price of bonds
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 repay  $B$  and set next period's asset level 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 it faces the choice of whether to pay or default at the beginning of a period
- $v_c(B, y)$  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 the probability of default next period given current holdings  $B'$  is

$$\delta(B', y) := \int \mathbb{1}\{v_c(B', y') < v_d(y')\} p(y, y') dy' \quad (3.152)$$

Given zero profits for foreign creditors in equilibrium, we can now pin down the bond price by combining (3.151) and (3.152) to obtain

$$q(B', y) = \frac{1 - \delta(B', y)}{1 + r} \quad (3.153)$$

### 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, 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

The code can be found in the file `arellano_vfi.jl` from the `QuantEcon` package but we repeat it here for convenience

(Results and discussion are given below the code)

```
using QuantEcon: tauchen, MarkovChain, simulate

# ----- #
# Define the main Arellano Economy type
# ----- #

"""
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.

##### Fields
* `β::Real`: Time discounting parameter
* `γ::Real`: Risk aversion parameter
* `r::Real`: World interest rate
* `ρ::Real`: Autoregressive coefficient on income process
* `η::Real`: Standard deviation of noise in income process
* `θ::Real`: Probability of re-entering the world financial sector after default
* `ny::Int`: Number of points to use in approximation of income process
* `nB::Int`: Number of points to use in approximation of asset holdings
* `ygrid::Vector{Float64}`: This is the grid used to approximate income process
* `ydefgrid::Vector{Float64}`: When in default get less income than process
    would otherwise dictate
* `Bgrid::Vector{Float64}`: This is grid used to approximate choices of asset
    holdings
* `Π::Array{Float64, 2}`: Transition probabilities between income levels
* `vf::Array{Float64, 2}`: Place to hold value function
* `vd::Array{Float64, 2}`: Place to hold value function when in default
* `vc::Array{Float64, 2}`: Place to hold value function when choosing to
    continue
* `policy::Array{Float64, 2}`: Place to hold asset policy function
* `q::Array{Float64, 2}`: Place to hold prices at different pairs of  $(y, B')$ 
* `defprob::Array{Float64, 2}`: Place to hold the default probabilities for
    pairs of  $(y, B')$ 
"""

immutable ArellanoEconomy
    # Model Parameters
    β::Float64
    γ::Float64
    r::Float64
    ρ::Float64
    η::Float64
    θ::Float64
```

```

# Grid Parameters
ny::Int
nB::Int
ygrid::Array{Float64, 1}
ydefgrid::Array{Float64, 1}
Bgrid::Array{Float64, 1}
Π::Array{Float64, 2}

# Value function
vf::Array{Float64, 2}
vd::Array{Float64, 2}
vc::Array{Float64, 2}
policy::Array{Float64, 2}
q::Array{Float64, 2}
defprob::Array{Float64, 2}
end

"""
This is the default constructor for building an economy as presented
in Arellano 2008.

##### Arguments
* `β::Real(0.953)`: Time discounting parameter
* `γ::Real(2.0)`: Risk aversion parameter
* `r::Real(0.017)`: World interest rate
* `ρ::Real(0.945)`: Autoregressive coefficient on income process
* `η::Real(0.025)`: Standard deviation of noise in income process
* `θ::Real(0.282)`: Probability of re-entering the world financial sector
  after default
* `ny::Int(21)`: Number of points to use in approximation of income process
* `nB::Int(251)`: Number of points to use in approximation of asset holdings
"""

function ArellanoEconomy(;β=.953, γ=2., r=0.017, ρ=0.945, η=0.025, θ=0.282,
                      ny=21, nB=251)

    # Create grids
    Bgrid = collect(linspace(-.4, .4, nB))
    ly, Π = tauchen(ny, ρ, η)
    ygrid = exp(ly)
    ydefgrid = min(.969 * mean(ygrid), ygrid)

    # Define value functions (Notice ordered different than Python to take
    # advantage of column major layout of Julia)
    vf = zeros(nB, ny)
    vd = zeros(1, ny)
    vc = zeros(nB, ny)
    policy = Array(Int, nB, ny)
    q = ones(nB, ny) .* (1 / (1 + r))
    defprob = Array(Float64, nB, ny)

    return ArellanoEconomy(β, γ, r, ρ, η, θ, ny, nB, ygrid, ydefgrid, Bgrid, Π,
                          vf, vd, vc, policy, q, defprob)
end

```

```

u(ae::ArellanoEconomy, c) = c^(1 - ae.γ) / (1 - ae.γ)
_unpack(ae::ArellanoEconomy) =
    ae.β, ae.γ, ae.r, ae.ρ, ae.η, ae.θ, ae.ny, ae.nB
_unpackgrids(ae::ArellanoEconomy) =
    ae.ygrid, ae.ydefgrid, ae.Bgrid, ae.Π, ae.vf, ae.vd, ae.vc, ae.policy, ae.q, ae.defprob

# -----
# Write the value function iteration
# -----
"""

This function performs the one step update of the value function for the
Arellano model-- Using current value functions and their expected value,
it updates the value function at every state by solving for the optimal
choice of savings

##### Arguments

* `ae::ArellanoEconomy`: This is the economy we would like to update the
  value functions for
* `EV::Matrix{Float64}`: Expected value function at each state
* `EVd::Matrix{Float64}`: Expected value function of default at each state
* `EVc::Matrix{Float64}`: Expected value function of continuing at each state

##### Notes

* This function updates value functions and policy functions in place.
"""

function one_step_update!(ae::ArellanoEconomy, EV::Matrix{Float64},
                           EVd::Matrix{Float64}, EVc::Matrix{Float64})

    # Unpack stuff
    β, γ, r, ρ, η, θ, ny, nB = _unpack(ae)
    ygrid, ydefgrid, Bgrid, Π, vf, vd, vc, policy, q, defprob = _unpackgrids(ae)
    zero_ind = searchsortedfirst(Bgrid, 0.)

    for iy=1:ny
        y = ae.ygrid[iy]
        ydef = ae.ydefgrid[iy]

        # Value of being in default with income y
        defval = u(ae, ydef) + β*(θ*EVc[zero_ind, iy] + (1-θ)*EVd[1, iy])
        ae.vd[1, iy] = defval

        for ib=1:nB
            B = ae.Bgrid[ib]

            current_max = -1e14
            pol_ind = 0
            for ib_next=1:nB
                c = max(y - ae.q[ib_next, iy]*Bgrid[ib_next] + B, 1e-14)
                m = u(ae, c) + β * EV[ib_next, iy]
                if m > current_max
                    current_max = m
                    pol_ind = ib_next
                end
            end
            ae.vc[ib, iy] = current_max
            ae.q[ib, iy] = pol_ind
        end
    end
end

```

```

        if m > current_max
            current_max = m
            pol_ind = ib_next
        end

    end

    # Update value and policy functions
    ae.vc[ib, iy] = current_max
    ae.policy[ib, iy] = pol_ind
    ae.vf[ib, iy] = defval > current_max ? defval: current_max
end
end

nothing
end

"""
This function takes the Arellano economy and its value functions and
policy functions and then updates the prices for each (y, B') pair

##### Arguments

* `ae::ArellanoEconomy`: This is the economy we would like to update the
  prices for

##### Notes

* This function updates the prices and default probabilities in place
"""

function compute_prices!(ae::ArellanoEconomy)
    # Unpack parameters
    β, γ, r, ρ, η, θ, ny, nB = _unpack(ae)

    # Create default values with a matching size
    vd_compat = repmat(ae.vd, nB)
    default_states = vd_compat .> ae.vc

    # Update default probabilities and prices
    copy!(ae.defprob, default_states * ae.Π')
    copy!(ae.q, (1 - ae.defprob) / (1 + r))

    nothing
end

"""
This performs value function iteration and stores all of the data inside
the ArellanoEconomy type.

##### Arguments

* `ae::ArellanoEconomy`: This is the economy we would like to solve
* `;tol::Float64(1e-8)": Level of tolerance we would like to achieve
"""

```

```

* `;maxit::Int(10000)`: Maximum number of iterations

##### Notes

* This updates all value functions, policy functions, and prices in place.

"""

function vfi!(ae::ArellanoEconomy; tol=1e-8, maxit=10000)

    # Unpack stuff
    β, γ, r, ρ, η, θ, ny, nB = _unpack(ae)
    ygrid, ydefgrid, Bgrid, Π, vf, vd, vc, policy, q, defprob = _unpackgrids(ae)
    Πt = Π'

    # Iteration stuff
    it = 0
    dist = 10.

    # Allocate memory for update
    V_upd = zeros(ae.vf)

    while dist > tol && it < maxit
        it += 1

        # Compute expectations for this iterations
        # (We need Π' because of order value function dimensions)
        copy!(V_upd, ae.vf)
        EV = ae.vf * Πt
        EVd = ae.vd * Πt
        EVc = ae.vc * Πt

        # Update Value Function
        one_step_update!(ae, EV, EVd, EVc)

        # Update prices
        compute_prices!(ae)

        dist = maxabs(V_upd - ae.vf)

        if it%25 == 0
            println("Finished iteration $(it) with dist of $(dist)")
        end
    end

    nothing
end

"""

This function simulates the Arellano economy

##### Arguments

* `ae::ArellanoEconomy`: This is the economy we would like to solve

```

```

* `capT::Int`: Number of periods to simulate
* `y_init::Float64(mean(ae.ygrid)` : The level of income we would like to
  start with
* `B_init::Float64(mean(ae.Bgrid)` : The level of asset holdings we would like
  to start with

##### Returns

* `B_sim_val::Vector{Float64}`: Simulated values of assets
* `y_sim_val::Vector{Float64}`: Simulated values of income
* `q_sim_val::Vector{Float64}`: Simulated values of prices
* `default_status::Vector{Float64}`: Simulated default status
  (true if in default)

##### Notes

* This updates all value functions, policy functions, and prices in place.

"""
function QuantEcon.simulate(ae::ArellanoEconomy, capT::Int=5000;
                             y_init=mean(ae.ygrid), B_init=mean(ae.Bgrid))

    # Get initial indices
    zero_index = searchsortedfirst(ae.Bgrid, 0.)
    y_init_ind = searchsortedfirst(ae.ygrid, y_init)
    B_init_ind = searchsortedfirst(ae.Bgrid, B_init)
    initial_dist, meany = zeros(ae.ny), mean(ae.ygrid)
    initial_dist[y_init_ind] = 1

    # Create a QE MarkovChain
    mc = MarkovChain(ae.Π)
    y_sim_indices = simulate(mc, initial_dist, capT+1)

    # Allocate and Fill output
    y_sim_val = Array(Float64, capT+1)
    B_sim_val, q_sim_val = similar(y_sim_val), similar(y_sim_val)
    B_sim_indices = Array(Int, capT+1)
    default_status = fill(false, capT+1)
    B_sim_indices[1], default_status[1] = B_init_ind, false
    y_sim_val[1], B_sim_val[1] = ae.ygrid[y_init_ind], ae.Bgrid[B_init_ind]

    for t=1:capT
        # Get today's indexes
        yi, Bi = y_sim_indices[t], B_sim_indices[t]
        defstat = default_status[t]

        # If you are not in default
        if !defstat
            default_today = ae.vc[Bi, yi] < ae.vd[yi] ? true: false

            if default_today
                # Default values
                default_status[t] = true
            end
        end
    end
end

```

```

        default_status[t+1] = true
        y_sim_val[t] = ae.ydefgrid[y_sim_indices[t]]
        B_sim_indices[t+1] = zero_index
        B_sim_val[t+1] = 0.
        q_sim_val[t] = ae.q[zero_index, y_sim_indices[t]]
    else
        default_status[t] = false
        y_sim_val[t] = ae.ygrid[y_sim_indices[t]]
        B_sim_indices[t+1] = ae.policy[Bi, yi]
        B_sim_val[t+1] = ae.Bgrid[B_sim_indices[t+1]]
        q_sim_val[t] = ae.q[B_sim_indices[t+1], y_sim_indices[t]]
    end

    # If you are in default
    else
        B_sim_indices[t+1] = zero_index
        B_sim_val[t+1] = 0.
        y_sim_val[t] = ae.ydefgrid[y_sim_indices[t]]
        q_sim_val[t] = ae.q[zero_index, y_sim_indices[t]]

        # With probability  $\theta$  exit default status
        if rand() < ae. $\theta$ 
            default_status[t+1] = false
        else
            default_status[t+1] = true
        end
    end
end

return (y_sim_val[1:capT], B_sim_val[1:capT], q_sim_val[1:capT],
        default_status[1:capT])
end

```

## Results

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

In what follows, all results are computed using the same parameter values used by Arellano

The values can be seen in the function *ArellanoEconomy* shown above

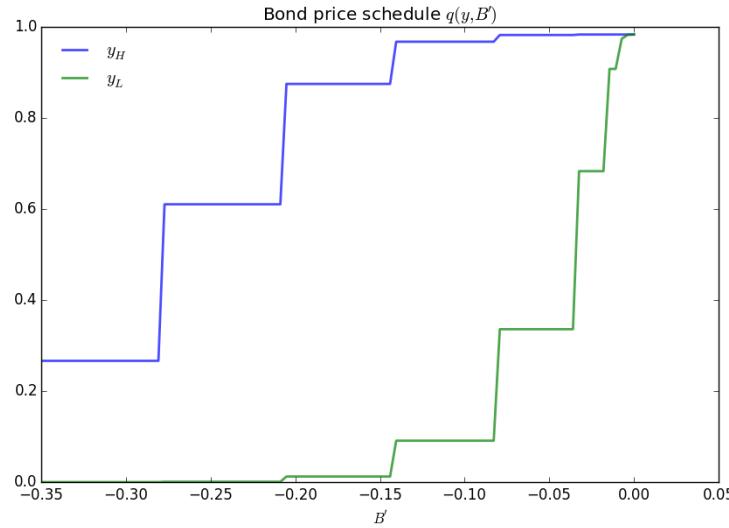
- For example,  $r=0.017$  matches the average quarterly rate on a 5 year US treasury over the period 1983–2001

Details on how to compute the figures are given as solutions to the exercises

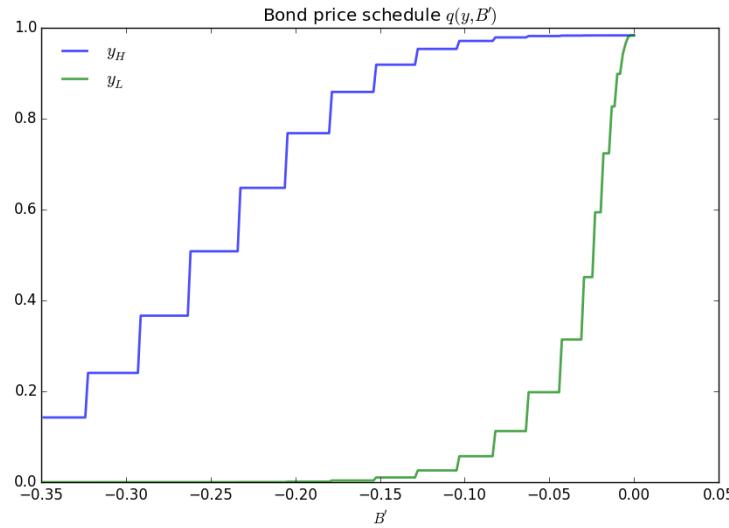
The first figure shows the bond price schedule and replicates Figure 3 of Arellano

- $y_L$  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 the original



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



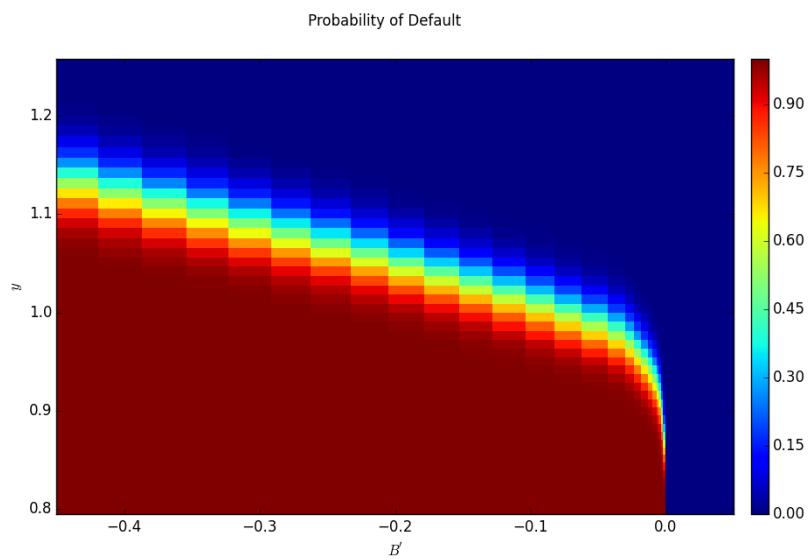
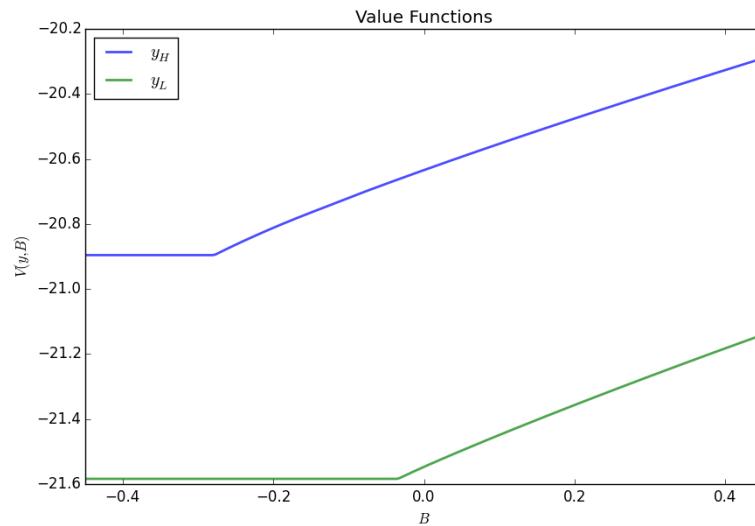
In either case, the figure shows that

- 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 (3.152)

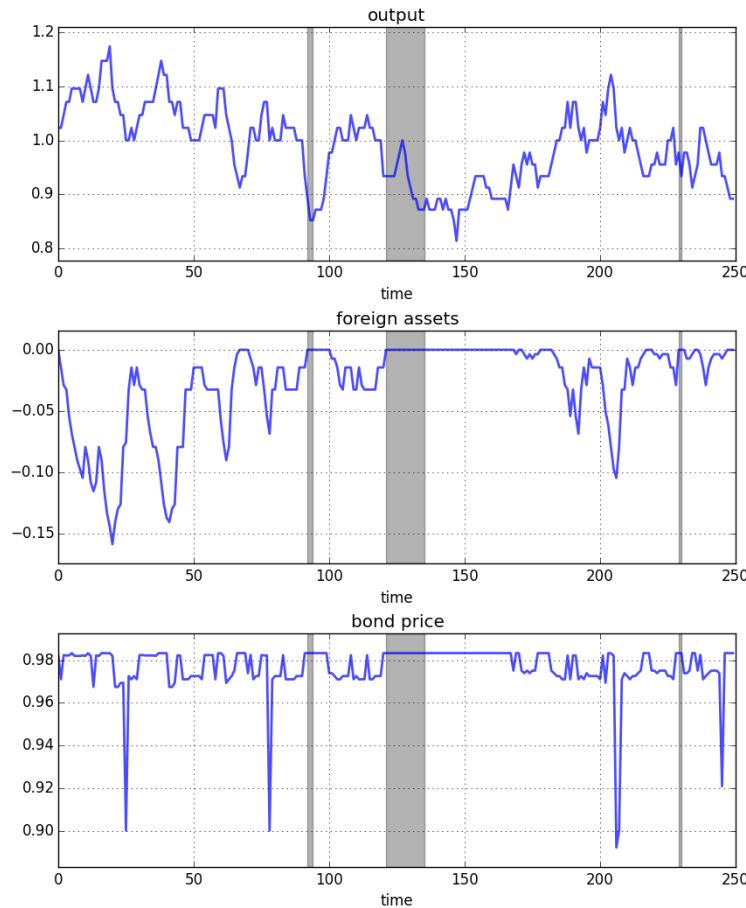
The next plot shows these default probabilities over  $(B', y)$  as a heat map



As anticipated, the probability that the government chooses to default in the following period increases with indebtedness and with 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 in 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 function *ArellanoEconomy*
- The time series will of course vary depending on the shock draws

**Solutions**

Solution notebook

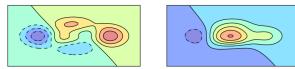
## REFERENCES

- [Aiy94] S Rao Aiyagari. Uninsured Idiosyncratic Risk and Aggregate Saving. *The Quarterly Journal of Economics*, 109(3):659–684, 1994.
- [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.
- [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.
- [Bis06] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
- [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.
- [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.
- [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.
- [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.
- [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.
- [HS00] Lars Peter Hansen and Thomas J Sargent. Wanting robustness in macroeconomics. *Manuscript, Department of Economics, Stanford University.*, 2000.
- [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.
- [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.
- [KP80] Finn E Kydland and Edward C Prescott. Dynamic optimal taxation, rational expectations and optimal control. *Journal of Economic Dynamics and Control*, 2:79–91, 1980.
- [LM94] A Lasota and M C MacKey. *Chaos, Fractals, and Noise: Stochastic Aspects of Dynamics*. Applied Mathematical Sciences. Springer-Verlag, 1994.
- [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.

- [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.
- [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.
- [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.
- [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.
- [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

AR, 362  
ARMA, 359, 362  
ARMA Processes, 356

### B

Bellman Equation, 332

### C

Central Limit Theorem, 131, 137  
Intuition, 138  
Multivariate Case, 141

CLT, 131

Complex Numbers, 360  
Continuous State Markov Chains, 263  
Covariance Stationary, 357  
Covariance Stationary Processes, 356  
AR, 358  
MA, 358

### D

Dynamic Programming, 185, 187  
Computation, 189  
Shortest Paths, 123  
Theory, 187  
Unbounded Utility, 188  
Value Function Iteration, 188, 189

### E

Eigenvalues, 87, 99  
Eigenvectors, 87, 99  
Ergodicity, 104, 116

### F

Finite Markov Asset Pricing, 237, 240  
Lucas Tree, 240  
Finite Markov Chains, 103–105  
Stochastic Matrices, 104

Fixed Point Theory, 282

### G

General Linear Processes, 358

### H

History Dependent Public Policies, 402  
Competitive Equilibrium, 404  
Ramsey Timing, 403  
Sequence of Governments Timing, 403  
Timing Protocols, 403

### I

Infinite Horizon Dynamic Programming, 185  
Irreducibility and Aperiodicity, 104, 110

### K

Kalman Filter, 166  
Programming Implementation, 172  
Recursive Procedure, 171

### L

Law of Large Numbers, 131  
Illustration, 133  
Multivariate Case, 141  
Proof, 132  
Linear Algebra, 87  
Differentiating Linear and Quadratic Forms, 102  
Eigenvalues, 99  
Eigenvectors, 99  
Matrices, 92  
Matrix Norms, 101  
Neumann’s Theorem, 101  
Positive Definite Matrices, 102  
Series Expansions, 101  
Spectral Radius, 102  
Vectors, 88

- Linear State Space Models, 144  
 Distributions, 150, 151  
 Ergodicity, 155  
 Martingale Difference Shocks, 146  
 Moments, 150  
 Moving Average Representations, 150  
 Prediction, 160  
 Seasonals, 148  
 Stationarity, 155  
 Time Trends, 149  
 Univariate Autoregressive Processes, 147  
 Vector Autoregressions, 148
- LLN, 131
- LQ Control, 201  
 Infinite Horizon, 212  
 Optimality (Finite Horizon), 205
- Lucas Model, 278  
 Assets, 279  
 Computation, 283  
 Consumers, 279  
 Dynamic Program, 280  
 Equilibrium Constraints, 281  
 Equilibrium Price Function, 281  
 Pricing, 279  
 Solving, 281
- M**
- Marginal Distributions, 104, 108  
 Markov Asset Pricing, 237  
 Overview, 237
- Markov Chains, 105  
 Calculating Stationary Distributions, 115  
 Continuous State, 263  
 Convergence to Stationarity, 116  
 Cross-Sectional Distributions, 110  
 Ergodicity, 116  
 Forecasting Future Values, 117  
 Future Probabilities, 110  
 Irreducibility, Aperiodicity, 110  
 Marginal Distributions, 108  
 Simulation, 106  
 Stationary Distributions, 114
- Matrix  
 Determinants, 97  
 Inverse, 97  
 Maps, 95  
 Operations, 93  
 Solving Systems of Equations, 95
- McCall Model, 308  
 Modeling  
 Career Choice, 287
- Models  
 Linear State Space, 146  
 Lucas Asset Pricing, 278  
 Markov Asset Pricing, 237  
 McCall, 307  
 On-the-Job Search, 296  
 Permanent Income, 247  
 Pricing, 237  
 Schelling's Segregation Model, 126
- N**
- Neumann's Theorem, 101  
 Nonparametric Estimation, 376
- O**
- On-the-Job Search, 296  
 Model, 297  
 Model Features, 297  
 Parameterization, 298  
 Programming Implementation, 299  
 Solving for Policies, 305
- Optimal Growth  
 Model, 186  
 Policy Function, 193  
 Policy Function Approach, 186
- Optimal Savings, 318  
 Computation, 320  
 Problem, 319  
 Programming Implementation, 322
- Optimal Taxation, 385
- P**
- Periodograms, 372  
 Computation, 374  
 Interpretation, 373
- Permanent Income Model, 247  
 Hall's Representation, 254  
 Savings Problem, 248
- Positive Definite Matrices, 102
- Pricing Models, 237  
 Finite Markov Asset Pricing, 240  
 Risk Aversion, 238  
 Risk Neutral, 237
- Programming  
 Dangers, 195

Iteration, 199  
 Writing Reusable Code, 195

Operations, 89  
 Span, 90

## R

Ramsey Problem, 402, 405  
 Computing, 406  
 Credible Policy, 421  
 Optimal Taxation, 385  
 Recursive Representation, 409  
 Time Inconsistency, 417  
 Two Subproblems, 407  
 Rational Expectations Equilibrium, 228  
 Competitive Equilibrium (w. Adjustment Costs), 230  
 Computation, 233  
 Definition, 230  
 Planning Problem Approach, 234  
 Robustness, 331

## W

White Noise, 357, 361  
 Wold's Decomposition, 358

## S

Schelling Segregation Model, 126  
 Smoothing, 372, 376  
 Spectra, 372  
 Estimation, 372  
 Spectra, Estimation  
 AR(1) Setting, 382  
 Fast Fourier Transform, 372  
 Pre-Filtering, 380  
 Smoothing, 376, 377, 380  
 Spectral Analysis, 356, 360  
 Spectral Densities, 361  
 Spectral Density, 362  
 interpretation, 362  
 Inverting the Transformation, 364  
 Mathematical Theory, 364  
 Spectral Radius, 102  
 Stationary Distributions, 104, 114  
 Stochastic Matrices, 104

## U

Unbounded Utility, 188

## V

Value Function Iteration, 188  
 Vectors, 87, 88  
 Inner Product, 90  
 Linear Independence, 92  
 Norm, 90