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

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

https://lectures.quantecon.org

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

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

1 Programming in Julia .......................... 1
  1.1 Setting up Your Julia Environment ...................... 1
  1.2 An Introductory Example .................................. 19
  1.3 Julia Essentials ....................................... 37
  1.4 Vectors, Arrays and Matrices ............................. 59
  1.5 Types, Methods and Dispatch ............................ 79
  1.6 The Need for Speed ................................... 96
  1.7 Plotting in Julia .................................... 110
  1.8 Useful Libraries .................................... 128

2 Tools and Techniques ......................... 141
  2.1 Linear Algebra ................................... 141
  2.2 Orthogonal Projections and Their Applications ....... 166
  2.3 LLN and CLT ....................................... 183
  2.4 Linear State Space Models ....................... 203
  2.5 Finite Markov Chains ................................ 229
  2.6 Continuous State Markov Chains ...................... 255
  2.7 A First Look at the Kalman Filter ..................... 278

3 Dynamic Programming ......................... 297
  3.1 Shortest Paths .................................. 297
  3.2 Job Search I: The McCall Search Model .............. 307
  3.3 Job Search II: Search and Separation ............... 320
  3.4 A Problem that Stumped Milton Friedman .......... 332
  3.5 Job Search III: Search with Learning ............... 354
  3.6 Job Search IV: Modeling Career Choice ............. 371
  3.7 Job Search V: On-the-Job Search ..................... 384
  3.8 Optimal Growth I: The Stochastic Optimal Growth Model .............................................. 398
  3.9 Optimal Growth II: Time Iteration ................. 419
  3.10 Optimal Growth III: The Endogenous Grid Method ........................................ 440
  3.11 LQ Dynamic Programming Problems ................... 450
  3.12 Optimal Savings I: The Permanent Income Model ........................................... 480
  3.13 Optimal Savings II: LQ Techniques .................... 497
  3.14 Consumption and Tax Smoothing with Complete and Incomplete Markets ......................... 514
  3.15 Optimal Savings III: Occasionally Binding Constraints ............................................ 533
3.16 Robustness ................................................................. 552
3.17 Discrete State Dynamic Programming ................................. 571

4 Multiple Agent Models .......................................................... 595
  4.1 Schellings Segregation Model ............................................. 595
  4.2 A Lake Model of Employment and Unemployment ................ 606
  4.3 Rational Expectations Equilibrium ..................................... 632
  4.4 Markov Perfect Equilibrium ............................................. 647
  4.5 Asset Pricing I: Finite State Models ................................. 664
  4.6 Asset Pricing II: The Lucas Asset Pricing Model .................. 683
  4.7 Asset Pricing III: Incomplete Markets ............................... 694
  4.8 Uncertainty Traps ...................................................... 705
  4.9 The Aiyagari Model .................................................... 719
  4.10 Default Risk and Income Fluctuations .............................. 728
  4.11 Globalization and Cycles ............................................. 752

5 Time Series Models ............................................................ 773
  5.1 Covariance Stationary Processes ...................................... 773
  5.2 Estimation of Spectra .................................................. 791
  5.3 Additive Functionals ................................................... 805
  5.4 Multiplicative Functionals ............................................. 824
  5.5 Classical Control with Linear Algebra ............................... 846
  5.6 Classical Filtering With Linear Algebra ............................ 870

6 Dynamic Programming Squared ............................................ 891
  6.1 Dynamic Stackelberg Problems ...................................... 891
  6.2 Optimal Taxation in an LQ Economy ................................. 907
  6.3 Optimal Taxation with State-Contingent Debt ..................... 928
  6.4 Optimal Taxation without State-Contingent Debt .................. 963

7 References ......................................................................... 995

8 Quantitative Economics ...................................................... 997

9 Tools and Techniques ......................................................... 999

10 Dynamic Programming .................................................... 1001

11 Multiple Agent Models ..................................................... 1003

12 Quantitative Economics .................................................... 1005

13 Tools and Techniques ....................................................... 1007

14 Dynamic Programming .................................................... 1009

15 Multiple Agent Models ..................................................... 1011

16 Time Series Models ........................................................ 1013
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
  - Installing Packages
  - Jupyter
  - Alternatives to Jupyter
  - Exercises

1.1.1 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
1.1.2 First Steps

Installation

To install Julia, get the current release from the download page

Note: In these lectures we assume you have version 0.6 or later

Unless you have good reason to do otherwise, choose

- The current release rather than nightly build
- 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)

The REPL

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

Lets try some basic commands
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

1.1. Setting up Your Julia Environment
You can also type `;` to get a shell prompt, at which you can enter shell commands

```
 julia> x = 10
  10
 julia> 2 * x
  20
```

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

Below well often show interactions with the interpreter as follows

```
x = 10
10
2 * x
20
```

### 1.1.3 Installing Packages

In these lectures youll often see statements such as

```
using Plots
```

or

```
using QuantEcon
```

These commands pull in code from some of Julias many external Julia code libraries

For the code to run, you need to install the corresponding package first
Fortunately this is easy using Julias package management system

For example, lets 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

Here's how it looks on our machine (which already has this package installed)

```
 julia> Pkg.add("DataFrames")
 INFO: Package DataFrames is already installed

 julia>
```

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

```julia
using DataFrames
```

Now its functions are accessible

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

```
2x2 DataFrame
<table>
<thead>
<tr>
<th>Row</th>
<th>x1</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>&quot;foo&quot;</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>&quot;bar&quot;</td>
</tr>
</tbody>
</table>
```
Keeping your Packages up to Date

Running

```
Pkg.update()
```

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

We assume throughout that you keep your packages updated to the latest version!

QuantEcon

QuantEcon is an organization that facilitates development of open source code for economic modeling

As well as these lectures, it supports QuantEcon.jl, a library for quantitative economic modeling in Julia

The installation method is standard

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

Here's an example, which creates a discrete approximation to an AR(1) process

```
using QuantEcon: tauchen

tauchen(4, 0.9, 1.0)
```

Discrete Markov Chain

stochastic matrix of type 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]
```

Well learn more about the library as we go along

1.1.4 Jupyter

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

One option that provides these features is Jupyter

As a bonus, Jupyter 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 Jupyter as your primary work environment or not, you'll find learning about it an excellent investment
Installing Jupyter

There are two steps here:

1. Installing Jupyter itself
2. Installing IJulia, which serves as an interface between Jupyter notebooks and Julia

In fact you can get both by installing IJulia

However, if you have the bandwidth, we recommend that you

1. Do the two steps separately
2. In the first step, when installing Jupyter, do this by installing the larger package Anaconda Python

The advantage of this approach is that Anaconda gives you not just Jupyter but the whole scientific Python ecosystem

This includes things like plotting tools well make use of later

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

Now open up a Julia terminal and type

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

If you have problems, consult the installation instructions

Other Requirements

Since IJulia runs in the browser it might be a good time 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 Started

Now either

1. search for and start the Jupyter notebook application on your machine or
2. open up a terminal (or `cmd` in Windows) and type `jupyter notebook`

You should see something (not exactly) like this

![Jupyter Notebook Dashboard](image)

The page you are looking at is called the dashboard

The address `localhost:8888/tree` you see in the image indicates that the browser is communicating with a Julia session via port 8888 of the local machine

If you click on New you should have the option to start a Julia notebook
Here's what your Julia notebook should look like.
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 Jupyter notebook is that it uses a modal editing system. This means that the effect of typing at the keyboard depends on which mode you are in. The two modes are:

1. Edit mode
   - Indicated by a green border around one cell, as in the pictures above
   - Whatever you type appears as is in that cell

2. Command mode
   - The green border is replaced by a blue border

1.1. Setting up Your Julia Environment
• Key strokes are interpreted as commands for example, typing b adds a new cell below the current one

(To learn about other commands available in command mode, go to Keyboard Shortcuts in the Help menu)

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 Jupyter notebook is a little tricky at first but very efficient when you get used to it

Working with Files

To run an existing Julia file using the notebook you can copy and paste the contents into a cell in the notebook
If its a long file, however, you have the alternative of
1. Saving the file in your present working directory
2. Executing include("filename") in a cell

The present working directory can be found by executing the command pwd()

Plots

Lets generate some plots

There are several options well discuss in detail later

For now lets start with Plots.jl

```julia
Pkg.add("Plots")
```

Now try copying the following into a notebook cell and hit Shift-Enter

```julia
using Plots
plot(sin, -2pi, pi, label="sine function")
```

Youll see something like this (although the style of plot depends on your installation more on this later)
Working with the Notebook

Let's go over some more Jupyter notebook features 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`.

1.1. Setting up Your Julia Environment
IJulia offers up the possible completions
This helps remind you of what's available and saves a bit of typing

**Online Help**

To get help on the Julia function such as `repmat`, enter `?repmat`

Documentation should now appear in the browser
Other Content

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

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

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

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

Now we Shift + Enter to produce this
Inserting unicode (e.g., Greek letters)

Julia supports the use of unicode characters such as $\alpha$ and $\beta$ in your code.

Unicode characters can be typed quickly in Jupyter using the `tab` key.

Try creating a new code cell and typing `\alpha`, then hitting the `tab` key on your keyboard.

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 current working directory.
These shell commands are handled by your default system shell and hence are platform specific

**Sharing Notebooks**

Notebook files are just text files structured in JSON and typically end 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 Jupyter notebook) and run the cells or edit as discussed above.

**nbviewer**

The Jupyter 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.

**1.1.5 Alternatives to Jupyter**

In this lecture series, we will assume that you are using Jupyter.

Doing so allows us to make sure that everything works in at least one sensible environment.

But as you work more with Julia, you will want to explore other environments as well.

Here are some notes on working with the REPL, text editors, and other alternatives.

**Editing Julia Scripts**

You can run Julia scripts from the REPL using the `include("filename")` syntax.

The file needs to be in the present working directory, which you can determine by typing `pwd()`.

You also need to know how to edit them; let's discuss how to do this without Jupyter.

**IDEs**

IDEs (Integrated Development Environments) combine an interpreter and text editing facilities in the one application.

For Julia, one nice option is Juno.
Text Editors

The beauty of text editors is that if you master one of them, you can use it for every coding task you come across, regardless of the language.

At a minimum, a text editor for coding should provide

- Syntax highlighting for the languages you want to work with
- Automatic indentation
- Efficient text manipulation (search and replace, copy and paste, etc.)

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

Suggestions:

- **Atom** is a popular open source next generation text editor
- **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**

1.1.6 Exercises

Exercise 1

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

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

This should start the kernel without launching the browser.

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

Now

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

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

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

Exercise 2

This exercise will familiarize you with git and GitHub.

**Git** is a *version control system* a piece of software used to manage digital projects such as code libraries.
In many cases the associated collections of files called repositories are stored on GitHub. GitHub is a wonderland of collaborative coding projects. Git is an extremely powerful tool for distributed collaboration: for example, we use it to share and synchronize all the source files for these lectures.

There are two main flavors of Git

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

As an exercise, try

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

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

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

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

Even better,

1. Sign up to GitHub
2. Look into forking GitHub repositories (forking means making your own copy of a GitHub repository, stored on GitHub)
3. Fork QuantEcon.jl
4. Clone your fork to some local directory, make edits, commit them, and push them back up to your forked GitHub repo
5. If you made a valuable improvement, send us a pull request!

For reading on these and other topics, try

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

1.2 An Introductory Example
1.2.1 Overview

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

Level

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, then one option is to try Python first. Python is a great first language and there are many introductory treatments. Otherwise, just dive in and see how you go.

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.

Set Up

We assume that you’ve worked your way through our getting started lecture already. For this lecture, we recommend that you work in a Jupyter notebook, as described here.
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

1.2.2 Example: Plotting a White Noise Process

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

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

This is straightforward using Plots.jl, which was discussed in our set up lecture

Fire up a Jupyter notebook and enter the following in a cell

```julia
using Plots
ts_length = 100
_values = randn(ts_length)
plot(_values, color="blue")
```

Let's break this down and see how it works
**Importing Functions**

The effect of the statement `using Plots` is to make all the names exported by the `Plots` module available in the global scope.

If you prefer to be more selective you can replace `using Plots` with `import Plots: 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 `_values = randn(ts_length)` creates one of the most fundamental Julia data types: an array.

```
typeof(_values)
Array{Float64,1}

_values
```

```
100-element Array{Float64,1}:
  -0.16706
  -0.44692
  0.0131702
  0.187735
  -1.84894
  0.729719
  -1.62587
  0.518996
  -0.723798
  0.00738428
  -0.394932
  0.0825062
  1.81214
  0.148999
  0.0376599
  1.20136
  1.47377
  0.951305
  0.100777
  0.189895
  0.883207
  1.21965
  -1.35306
  -1.14473
  0.527658
```
The information from `typeof()` tells us that `_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:

```
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:

```
typeof(x[1])
```

Int64

The second is a string:

```
typeof(x[2])
```

String

The third is the boolean value `false`:

```
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 are several examples, applied to the same list `x = [10, "foo", false]`:

```
length(x)
```

3

```
pop!(x)
```

false
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**.

Its 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, except that it appends its second argument to the array.

**For Loops**

Although there is no need in terms of what we wanted to achieve with our program, for the sake of learning syntax lets rewrite our program to use a `for` loop:

```julia
# ts_length = 100
_values = Array{Float64}(ts_length)
for i in 1:ts_length
    _values[i] = randn()
end
plot(_values, color="blue")
```
Here we first declared `_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:

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

Hello foo
Hello bar

**While Loops**

The syntax for the while loop contains no surprises.

```julia
ts_length = 100
_values = Array{Float64}(ts_length)
```

1.2. An Introductory Example
\begin{verbatim}

i = 1
while i <= ts_length
    _values[i] = randn()
    i = i + 1
end
plot(_values, color="blue")

\end{verbatim}

The next example does the same thing with a condition and the \texttt{break} statement

\begin{verbatim}

ts_length = 100
_values = Array\{Float64\}(ts_length)
i = 1
while true
    _values[i] = randn()
    i = i + 1
    if i > ts_length
        break
    end
end
plot(_values, color="blue")

\end{verbatim}
User-Defined Functions

For the sake of the exercise, let’s now go back to the for loop but restructure our program so that generation of random variables takes place within a user-defined function

```julia
function generate_data(n)
    _values = Array(Float64)(n)
    for i = 1:n
        _values[i] = randn()
    end
    return _values
end

ts_length = 100
data = generate_data(ts_length)
plot(data, color="blue")
```

1.2. An Introductory Example
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

```julia
ts_length = 100
data = randn(ts_length)
plot(data, color="blue")
```
Lets 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 well make use of the Distributions package

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

Heres the code

```julia
using Distributions

function plot_histogram(distribution, n)
    _values = rand(distribution, n)  # n draws from distribution
    histogram(_values)
end

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

The resulting figure looks like this
Lets have a casual discussion of how all this works while leaving technical details for later in the lectures

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

The name \( \text{lp} \) is bound to this object

When we make the function call \( \text{plot}\_\text{histogram}(\text{lp}, 500) \) the code in the body of the function \( \text{plot}\_\text{histogram} \) is run with

- the name \( \text{distribution} \) bound to the same object as \( \text{lp} \)
- the name \( n \) bound to the integer 500

**A Mystery**

Now consider the function call \( \text{rand}(\text{distribution}, n) \)

This looks like something of a mystery

The function \( \text{rand}() \) is defined in the base library such that \( \text{rand}(n) \) returns \( n \) uniform random variables on \([0,1)\)

```julia
rand(3)
```

3-element Array{Float64,1}:
0.282284
0.701812
0.512708
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**

### 1.2.3 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 Julias 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 $U_1, \ldots, U_n$ are iid copies of $U$, then, as $n$ gets large, the fraction that falls in $B$ converges to the probability of landing in $B$

• For a circle, area = $\pi \cdot \text{radius}^2$

**Exercise 4**

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

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

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

**Exercise 5**

Simulate and plot the correlated time series

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

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

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

**Exercise 6**

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)

### 1.2.4 Solutions

#### Exercise 1

```julia
function factorial2(n)
    k = 1
    for i in 1:n
        k = k * i
    end
    return k
end

factorial2(4)
```

```
24
```

```julia
factorial(4)  # Built-in function
```

```
24
```
Exercise 2

```julia
function binomial_rv(n, p)
    count = 0
    U = rand(n)
    for i in 1:n
        if U[i] < p
            count = count + 1  # Or count += 1
        end
    end
    return count
end
for j in 1:25
    b = binomial_rv(10, 0.5)
    print("\$b, ")
end
```

Exercise 3

Consider the circle of diameter 1 embedded in the unit square

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

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

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

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

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

```julia
n = 1000000
count = 0
for i in 1:n
    u, v = rand(2)
    d = sqrt((u - 0.5)^2 + (v - 0.5)^2)  # Distance from middle of square
    if d < 0.5
        count += 1
    end
end
area_estimate = count / n
print(area_estimate * 4)  # dividing by radius**2
```

3.139596
Exercise 4

```julia
payoff = 0
count = 0

print("Count = ")
for i in 1:10
    U = rand()
    if U < 0.5
        count += 1
    else
        count = 0
    end
    print(count)
    if count == 3
        payoff = 1
    end
end

println("payoff = ")
println("payoff = ")
```

Count = 1010010120
payoff = 0

We can simplify this somewhat using the ternary operator. Here's some examples

```julia
a = 1 < 2 ? "foo" : "bar"
a

"foo"

a = 1 > 2 ? "foo" : "bar"
a

"bar"
```

Using this construction:

```julia
payoff = 0
count = 0

print("Count = ")
for i in 1:10
    U = rand()
    count = U < 0.5 ? count + 1 : 0
    print(count)
    if count == 3
        payoff = 1
    end
end
```

1.2. An Introductory Example
end
print("\n")
println("payoff = ", payoff)

Count = 0010101230
payoff = 1

Exercise 5

Here is one solution

\[ \alpha = 0.9 \]
\[ T = 200 \]
\[ x = \text{zeros}(T + 1) \]

```julia
for t in 1:T
    x[t+1] = \alpha \times x[t] + \text{randn()}
end
```

Exercise 6

\[ \alpha = [0.0, 0.8, 0.98] \]
\[ T = 200 \]
```julia
series = []
labels = []

for α in αs
    x = zeros(T + 1)
    x[1] = 0
    for t in 1:T
        x[t+1] = α * x[t] + randn()
    end
    push!(series, x)
    push!(labels, "α = \$α")
end

plot(series, label=reshape(labels, 1, length(labels)))
```

1.3 Julia Essentials

Contents

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

### 1.3.1 Overview

Topics:

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

### 1.3.2 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`.

\[
\times = \text{true}
\]
true

typeof(x)

Bool

y = 1 > 2  # Now y = false

false

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

true + false

1

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)

typeof(1.0)

Float64

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

x = 2; y = 1.0

1.0

x + y

2.0

x^2
Although the \* can be omitted for multiplication between a numeric literal and a variable

\[
2x - 3y
\]

1.0

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

\[(10, 20)\]

30

\[(10, 20)\]

200

Complex numbers are another primitive data type, with the imaginary part being specified by \texttt{im}

\[
x = 1 + 2im
\]

1 + 2im

\[
y = 1 - 2im
\]

1 - 2im

\[
x \times y \ # Complex multiplication
\]

5 + 0im

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

**Strings**

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

\[
x = "foobar"
\]

"foobar"
typeof(x)  
String

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

```julia
x = 10; y = 20

20

"x = \$x"

"x = 10"

"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"

split(s)

3-element Array{SubString{String},1}:

  "Charlie"
  "don't"
  "surf"

replace(s, "surf", "ski")

"Charlie don't ski"

split("fee,fi,fo", ",")

3-element Array{SubString{String},1}:

  "fee"
  "fi"
  "fo"
```
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 digits in string
```

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")

typeof(x)

Tuple{String,String}
```

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

MethodError: no method matching setindex!(::Tuple{String,String}, ::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")
```
Referencing Items

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

```plaintext
x = [10, 20, 30, 40]
```

4-element Array{Int64,1}:
10
20
30
40

```plaintext
x[end]
```
40

```plaintext
x[end-1]
```
30

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

```plaintext
x[1:3]
```

3-element Array{Int64,1}:
10
20
30

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

The same slice notation works on strings

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

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

The strings \texttt{name} and \texttt{age} are called the \texttt{keys}

The objects that the keys are mapped to ("Frodo" and 33) are called the \texttt{values}

They can be accessed via \texttt{keys(d)} and \texttt{values(d)} respectively

**1.3.3 Input and Output**

Lets have a quick look at reading from and writing to text files

Well start with writing

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

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

8
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>)
print(readstring(f))
testing
more testing
```

Often when reading from a file we want to step through the lines of a file, performing an action on each one. Theres a neat interface to this in Julia, which takes us to our next topic.

### 1.3.4 Iterating

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

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

```
Charlie don't surf
Charlie don't ski
```
They also include so-called iterators
Youve already come across these types of objects

```
for i in 1:3 print(i) end
```

123

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

```
d = Dict("name" => "Frodo", "age" => 33)
```

```
Dict(String,Any) with 2 entries:
   "name" => "Frodo"
   "age" => 33
``` 

```
keys(d)
```

```
Base.KeyIterator for a Dict(String,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()`

```
collect(keys(d))
```

```
2-element Array{String,1}:
   "name"
   "age"
``` 

### Looping without Indices

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

For example compare

```
x_values = linspace(0, 3, 10)
```

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

```
0.0
0.1111111111111111
0.4444444444444444
1.0
```
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:

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

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

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:

```julia
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
```

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

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

Here's some examples

doubles = [2i for i in 1:4]

4-element Array{Int64,1}:
2
4
6
8

animals = ["dog", "cat", "bird"];  # Semicolon suppresses output

plurals = [animal * "s" for animal in animals]

3-element Array{String,1}:
"dogs"
"cats"
"birds"

[i + j for i in 1:3, j in 4:6]

3-element Array{Int64,2}:
5 6 7
6 7 8
7 8 9

[i + j + k for i in 1:3, j in 4:6, k in 7:9]

3-element Array{Int64,3}:
[:, :, 1] =
12 13 14
13 14 15
14 15 16

[:, :, 2] =
13 14 15
14 15 16
15 16 17

[:, :, 3] =
14 15 16
15 16 17
16 17 18

The same kind of expression works for dictionaries
Dict("$i" => i for i in 1:3)

Dict(String, Int64) with 3 entries:
  "1" => 1
  "2" => 2
  "3" => 3

1.3.5 Comparisons and Logical Operators

Comparisons

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

```julia
x = 1

1

x == 2
false
```

For not equal use !=

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

We can chain inequalities:

```julia
1 < 2 < 3
true

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: TypeError: non-boolean (Int64) used in boolean context
Stacktrace:
  [1] anonymous at ./<missing>:?

1.3. Julia Essentials
if 1 print("foo") end

TypeError: non-boolean (Int64) used in boolean context

Combining Expressions

Here are the standard logical connectives (conjunction, disjunction)

true && false
false

false

true || false
true

Remember

• P && Q is true if both are true, otherwise its false
• P || Q is false if both are false, otherwise its true

1.3.6 User-Defined Functions

Lets 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

Well see many examples of these structures in the following lectures

For now lets 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:

```julia
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
ff(x) = sin(1 / x)
```

Let's check that it works:

```julia
ff(1 / pi)
```

```
1.2246467991473532e-16
```

Julia also allows for you to define anonymous functions.

For example, to define \( f(x) = \sin\left(\frac{1}{x}\right) \) you can use \( x \rightarrow \sin\left(\frac{1}{x}\right) \).

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

How can you use a function with no name?

Typically its as an argument to another function:

```julia
map(x -> sin(1 / x), randn(3))  # Apply function to each element
```
Optional and Keyword Arguments

Function arguments can be given default values

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

If the argument is not supplied the default value is substituted

```julia
fff(pi)
```

0.36787944117144233

```julia
fff(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

```julia
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

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

1.3.7 Vectorized Functions

A common scenario in computing is that

- we have a function $f$ such that $f(x)$ returns a number for any number $x$
- we wish to apply $f$ to every element of a vector $x\_vec$ to produce a new vector $y\_vec$
In Julia loops are fast and we can do this easily enough with a loop

For example, suppose that we want to apply \( \sin \) to \( x_{\text{vec}} = [2.0, 4.0, 6.0, 8.0] \)

The following code will do the job

```julia
x_vec = [2.0, 4.0, 6.0, 8.0]
y_vec = similar(x_vec)
for (i, x) in enumerate(x_vec)
    y_vec[i] = sin(x)
end
```

But this is a bit unwieldy so Julia offers the alternative syntax

```julia
y_vec = sin.(x_vec)
```

More generally, if \( f \) is any Julia function, then \( f. \) references the vectorized version

Conveniently, this applies to user-defined functions as well

To illustrate, let’s write a function \( \text{chisq} \) such that \( \text{chisq}(k) \) returns a chi-squared random variable with \( k \) degrees of freedom when \( k \) is an integer

In doing this we exploit the fact that, if we take \( k \) independent standard normals, square them all and sum, we get a chi-squared with \( k \) degrees of freedom

```julia
function chisq(k::Integer)
    @assert k > 0 "k must be a natural number"
    z = randn(k)
    return sum(z.^2)
end

chisq(3)
```

```julia
1.5841392760511817
```

Note that calls with integers less than 1 will trigger an assertion failure inside the function body

```julia
chisq(-2)
```

```julia
AssertionError: k must be a natural number
```

Let’s try this out on an array of integers, adding the vectorized notation

```julia
chisq.([2, 4, 6])
```

3-element Array{Float64,1}:

```
0.992351
3.03434
3.29578
```
1.3.8 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 between 0 and 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

\[ p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \sum_{i=0}^{n} a_i x^i \]  \hspace{1cm} (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 \( a \leq x \leq b \)

and returns the piecewise linear interpolation of \( f \) at \( x \), based on \( n \) evenly spaced grid points \( a = \text{point}[1] < \text{point}[2] < \ldots < \text{point}[n] = b \)

Aim for clarity, not efficiency

Exercise 6

The following data lists US cities and their populations

<table>
<thead>
<tr>
<th>City</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>new york</td>
<td>8244910</td>
</tr>
<tr>
<td>los angeles</td>
<td>3819702</td>
</tr>
<tr>
<td>chicago</td>
<td>2707120</td>
</tr>
<tr>
<td>houston</td>
<td>2145146</td>
</tr>
<tr>
<td>philadelphia</td>
<td>1536471</td>
</tr>
<tr>
<td>phoenix</td>
<td>1469471</td>
</tr>
<tr>
<td>san antonio</td>
<td>1359758</td>
</tr>
<tr>
<td>san diego</td>
<td>1326179</td>
</tr>
<tr>
<td>dallas</td>
<td>1223229</td>
</tr>
</tbody>
</table>

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()`

This can also be achieved by running the following Julia code:

```julia
open("us_cities.txt", "w") do f
    write(f, "new york: 8244910
          los angeles: 3819702
          chicago: 2707120
          houston: 2145146
          philadelphia: 1536471
          phoenix: 1469471
          san antonio: 1359758
          san diego: 1326179
          dallas: 1223229")
end
```

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
• `parse(Int, "100")` converts the string "100" into an integer
1.3.9 Solutions

Exercise 1

Part 1 solution:

Heres one possible solution

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

6

Part 2 solution:

One solution is

```julia
sum([x % 2 == 0 for x in 0:99])
```

50

This also works

```julia
sum(map(x -> x % 2 == 0, 0:99))
```

50

Part 3 solution:

Heres one possibility

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

2

Exercise 2

```julia
p(x, coeff) = sum([a * x^(i-1) for (i, a) in enumerate(coeff)])
```

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

6

Exercise 3

Heres one solutions:
```julia
function f_ex3(string)
    count = 0
    for letter in string
        if (letter == uppercase(letter)) && isalpha(letter)
            count += 1
        end
    end
    return count
end

f_ex3("The Rain in Spain")
```

```julia
3
```

**Exercise 4**

Here's one solution:

```julia
function f_ex4(seq_a, seq_b)
    is_subset = true
    for a in seq_a
        if !(a in seq_b)
            is_subset = false
        end
    end
    return is_subset
end

# == test == #
println(f_ex4([1, 2], [1, 2, 3]))
println(f_ex4([1, 2, 3], [1, 2]))

true
false

if we use the `Set` data type then the solution is easier

```julia
f_ex4_2(seq_a, seq_b) = issubset(Set(seq_a), Set(seq_b))

println(f_ex4_2([1, 2], [1, 2, 3]))
println(f_ex4_2([1, 2, 3], [1, 2]))

true
false
```
Exercise 5

```julia
function linapprox(f, a, b, n, x)
    # Evaluates the piecewise linear interpolant of f at x on the interval [a, b], with n evenly spaced grid points.
    #
    length_of_interval = b - a
    num_subintervals = n - 1
    step = length_of_interval / num_subintervals
    # === find first grid point larger than x === #
    point = a
    while point <= x
        point += step
    end
    # === x must lie between the gridpoints (point - step) and point === #
    u, v = point - step, point
    return f(u) + (x - u) * (f(v) - f(u)) / (v - u)
end
```

Let's test it

```julia
f_ex5(x) = x^2
g_ex5(x) = linapprox(f_ex5, -1, 1, 3, x)
```

```julia
using Plots
pyplot()
x_grid = linspace(-1, 1, 100)
y_vals = map(f_ex5, x_grid)
y_approx = map(g_ex5, x_grid)
plot(x_grid, y_vals, label="true")
plot!(x_grid, y_approx, label="approximation")
```
Exercise 6

```julia
f_ex6 = open("us_cities.txt", "r")
total_pop = 0
for line in eachline(f_ex6)
    city, population = split(line, ':')  # Tuple unpacking
    total_pop += parse(Int, population)
end
close(f_ex6)
println("Total population = ", total_pop)
```

```
Total population = 23831986
```

1.4 Vectors, Arrays and Matrices

Contents

- Vectors, Arrays and Matrices
  - Overview
  - Array Basics
Lets 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

### 1.4.1 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

### 1.4.2 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(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: the size along the one dimension that exists.

Here are some functions that create two-dimensional arrays.

```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])
```

```julia
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.
Array(Int64, 1) == Vector(Int64)

true

Array(Int64, 2) == Matrix(Int64)

true

Array(Int64, 1) == Matrix(Int64)

false

Array(Int64, 3) == Matrix(Int64)

false

In particular, a Vector in Julia is a flat array

**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
b = reshape(a, 2, 2)

2×2 Array{Int64,2}:
10  30
20  40
```

Notice that this function returns a view on the existing array

This means that changing the data in the new array will modify the data in the old one:
b[1, 1] = 100  # Continuing the previous example

100

b

2×2 Array{Int64,2}:
100  30
20   40

a

4-element Array{Int64,1}:
100
20
30
40

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

a = [1 2 3 4]  # Two dimensional

1×4 Array{Int64,2}:
1 2 3 4

squeeze(a, 1)

4-element Array{Int64,1}:
1
2
3
4

The return value is an array with the specified dimension flattened

**Why Flat Arrays?**

As we've seen, in Julia we have both

- one-dimensional arrays (i.e., flat arrays)
- arrays of size (1, n) or (n, 1) that represent row and column vectors respectively

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
On the other, we use arrays in many settings that dont involve matrix algebra
In such cases, we dont care about the distinction between row and column vectors
This is why many Julia functions return flat arrays by default

**Creating Arrays**

**Functions that Return Arrays**

Weve already seen some functions for creating arrays

```plaintext
eye(2)
```

```
2x2 Array{Float64,2}:
  1.0  0.0
  0.0  1.0
```

```plaintext
zeros(3)
```

```
3-element Array{Float64,1}:
  0.0
  0.0
  0.0
```

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

```plaintext
x = Array{Float64}(2, 2)
```

```
2x2 Array{Float64,2}:
  6.92478e-310  6.92477e-310
  6.92477e-310  0.0
```

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

```plaintext
ones(2, 2)
```

```
2x2 Array{Float64,2}:
  1.0  1.0
  1.0  1.0
```

```plaintext
fill("foo", 2, 2)
```

```
2x2 Array{String,2}:
  "foo"  "foo"
  "foo"  "foo"
```
Manual Array Definitions

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

\[
a = [10, 20, 30, 40]
\]

4-element Array(Int64,1):
10
20
30
40

In two dimensions we can proceed as follows

\[
a = [10 20 30 40] \quad \text{# Two dimensional, shape is 1 x n}
\]

1Œ4 Array(Int64,2):
10 20 30 40

\[
\text{ndims}(a)
\]

2

\[
a = [10 20; 30 40] \quad \text{# 2 x 2}
\]

2Œ2 Array(Int64,2):
10 20
30 40

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

\[
a = [10; 20; 30; 40]
\]

4-element Array(Int64,1):
10
20
30
40

\[
\text{ndims}(a)
\]

1

Instead transpose the row vector

\[
a = [10 20 30 40]'
\]
4×1 Array{Int64,2}:
 10
 20
 30
 40

ndims(a)

2

Array Indexing

Weve 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)
```

2×2 Array{Float64,2}:
 0.665276  -3.12382
 1.49257   -0.756258

```julia
a[1, 1]
```

0.6652764178803451
Booleans can be used to extract elements

```julia
b = [true false; false true]
```

This is useful for conditional extraction, as well see below

An aside: some or all elements of an array can be set equal to one number using slice notation
a[2:end] = 42

42

a

4-element Array{Float64,1}:
  6.92478e-310
  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

a = ones(3)

3-element Array{Float64,1}:
  1.0
  1.0
  1.0

b = a

3-element Array{Float64,1}:
  1.0
  1.0
  1.0

b[3] = 44

44

a

3-element Array{Float64,1}:
  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, its very inefficient to copy arrays unnecessarily
If you do need an actual copy in Julia, just use `copy()`

\[
a = \text{ones}(3)
\]

3-element Array\{Float64,1\}:
1.0
1.0
1.0

\[
b = \text{copy}(a)
\]

3-element Array\{Float64,1\}:
1.0
1.0
1.0

\[
b[3] = 44
\]

44

\[
a
\]

3-element Array\{Float64,1\}:
1.0
1.0
1.0

1.4.3 Operations on Arrays

Array Methods

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

\[
a = [-1, 0, 1]
\]

3-element Array\{Int64,1\}:
-1
0
1

\[
\text{length}(a)
\]

3

\[
\text{sum}(a)
\]
0

mean(a)

0.0

std(a)

1.0

var(a)

1.0

maximum(a)

1

minimum(a)

-1

b = sort(a, rev=true)  # Returns new array, original not modified

3-element Array{Int64,1}:
  1
  0
  -1

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

false

b = sort!(a, rev=true)  # Returns *modified original* array

3-element Array{Int64,1}:
  1
  0
  -1

b === a

true
Matrix Algebra

For two dimensional arrays, \( \ast \) means matrix multiplication

\[
a = \text{ones}(1, 2)
\]

1\times2 Array\{Float64, 2\}:

\[
\begin{pmatrix}
1.0 \\
1.0
\end{pmatrix}
\]

\[
b = \text{ones}(2, 2)
\]

2\times2 Array\{Float64, 2\}:

\[
\begin{pmatrix}
1.0 & 1.0 \\
1.0 & 1.0
\end{pmatrix}
\]

\[
a \ast b
\]

1\times2 Array\{Float64, 2\}:

\[
\begin{pmatrix}
2.0 & 2.0
\end{pmatrix}
\]

\[
b \ast a'
\]

2\times1 Array\{Float64, 2\}:

\[
\begin{pmatrix}
2.0 \\
2.0
\end{pmatrix}
\]

To solve the linear system \( AX = B \) for \( X \) use \( A \backslash B \)

\[
A = [1 \ 2; 2 \ 3]
\]

2\times2 Array\{Int64, 2\}:

\[
\begin{pmatrix}
1 & 2 \\
2 & 3
\end{pmatrix}
\]

\[
B = \text{ones}(2, 2)
\]

2\times2 Array\{Float64, 2\}:

\[
\begin{pmatrix}
1.0 & 1.0 \\
1.0 & 1.0
\end{pmatrix}
\]

\[
A \backslash B
\]

2\times2 Array\{Float64, 2\}:

\[
\begin{pmatrix}
-1.0 & -1.0 \\
1.0 & 1.0
\end{pmatrix}
\]
\[ \text{inv}(A) \ast B \]

\[
2\times2 \text{ Array} \{\text{Float64},2\}:
\begin{pmatrix}
-1.0 & -1.0 \\
 1.0 &  1.0
\end{pmatrix}
\]

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:

\[
\text{ones}(2) \ast \text{ones}(2)
\]

MethodError: no method matching \( \ast(\cdot) \)

Closest candidates are:

\[
\ast(\cdot, \cdot) \text{ at operators.jl:138}
\]

If you want an inner product in this setting use \texttt{dot()}:

\[
\text{dot}(\text{ones}(2), \text{ones}(2))
\]

\[ 2.0 \]

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

\[
\text{b} = \text{ones}(2, 2)
\]

\[
2\times2 \text{ Array} \{\text{Float64},2\}:
\begin{pmatrix}
1.0 &  1.0 \\
1.0 &  1.0
\end{pmatrix}
\]
b * ones(2)

2-element Array{Float64,1}:
  2.0
  2.0

ones(2) * b

DimensionMismatch("A has dimensions (2,1) but B has dimensions (2,2)")
in gemm_wrapper!(::Array{Float64,2}, ::Char, ::Char, ::Array{Float64,2}, · · · , ::Array{Float64,2}) at ./linalg/matmul.jl:309

in *(::Array{Float64,1}, ::Array{Float64,2}) at ./linalg/matmul.jl:86

Its 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

\[
\text{ones}(2, 2) \ast \text{ones}(2, 2) \quad \# \text{Matrix multiplication}
\]

\[
\begin{array}{cc}
2.0 & 2.0 \\
2.0 & 2.0
\end{array}
\]

\[
\text{ones}(2, 2) \ast \text{ones}(2, 2) \quad \# \text{Element by element multiplication}
\]

\[
\begin{array}{cc}
1.0 & 1.0 \\
1.0 & 1.0
\end{array}
\]

This is a general principle: \( .\times \) means apply operator \( \times \) elementwise.

\[
A = -\text{ones}(2, 2)
\]

\[
\begin{array}{cc}
-1.0 & -1.0 \\
-1.0 & -1.0
\end{array}
\]
A.^2  # Square every element

2×2 Array{Float64,2}:
  1.0  1.0
  1.0  1.0

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

ones(2, 2) + ones(2, 2)  # Same as ones(2, 2) .+ ones(2, 2)

2×2 Array{Float64,2}:
  2.0  2.0
  2.0  2.0

Scalar multiplication is similar:

A = ones(2, 2)

2×2 Array{Float64,2}:
  1.0  1.0
  1.0  1.0

2 * A  # Same as 2 .* A

2×2 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 . style notation:

a = [10, 20, 30]

3-element Array{Int64,1}:
  10
  20
  30

b = [-100, 0, 100]

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

3-element BitArray{1}:
false
false
true

a .== b

3-element BitArray{1}:
false
false
false
false

We can also do comparisons against scalars with parallel syntax

b

3-element Array{Int64,1}:
-100
 0
100

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

a = randn(4)

4-element Array{Float64,1}:
-1.55242
-1.0166
-1.27692
 1.38091

a .< 0

4-element BitArray{1}:
true
true
true
false

1.4. Vectors, Arrays and Matrices
Vectorized Functions

Julia provides standard mathematical functions such as \texttt{log}, \texttt{exp}, \texttt{sin}, etc.

\begin{verbatim}
log(1.0)
\end{verbatim}

0.0

By default, these functions act \textit{elementwise} on arrays

\begin{verbatim}
log.(ones(4))
\end{verbatim}

4-element Array\{Float64,1\}:
  0.0
  0.0
  0.0
  0.0

Functions that act elementwise on arrays in this manner are called \textit{vectorized functions}

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

\begin{verbatim}
[log(x) for x in ones(4)]
\end{verbatim}

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

1.4.4 Linear Algebra

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

\begin{verbatim}
A = [1 2; 3 4]
\end{verbatim}
2×2 Array{Int64,2}:
1 2
3 4

\[ \text{det}(A) \]
-2.0

\[ \text{trace}(A) \]
5

\[ \text{eigvals}(A) \]
2-element Array{Float64,1}:
-0.372281
5.37228

\[ \text{rank}(A) \]
2

For more details see the linear algebra section of the standard library

### 1.4.5 Exercises

**Exercise 1**

This exercise is on some matrix operations that arise in certain problems, including when dealing with linear stochastic difference equations.

If you arent familiar with all the terminology dont be concerned you can skim read the background discussion and focus purely on the matrix exercise.

With that said, consider the stochastic difference equation

\[ X_{t+1} = AX_t + b + \Sigma W_{t+1} \]  \hspace{1cm} (1.2)

Here

- \( X_t, b \) and \( X_{t+1} \) are \( n \times 1 \)
- \( A \) is \( n \times n \)
- \( \Sigma \) is \( n \times k \)
- \( W_t \) is \( k \times 1 \) and \( \{ W_t \} \) is iid with zero mean and variance-covariance matrix equal to the identity matrix
Let \( S_t \) denote the \( n \times n \) variance-covariance matrix of \( X_t \).

Using the rules for computing variances in matrix expressions, it can be shown from (1.2) that \( \{ S_t \} \) obeys

\[
S_{t+1} = AS_tA' + \Sigma \Sigma'
\]

(1.3)

It can be shown that, provided 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

\[
S = ASA' + Q \quad \text{where} \quad Q := \Sigma \Sigma'
\]

(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

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

1.4.6 Solutions

Exercise 1

Here’s the iterative approach:

```julia
function compute_asymptotic_var(A,
    Sigma,
    S0=Sigma * Sigma',
    tolerance=1e-6,
    maxiter=500)

    V = Sigma * Sigma'
    S = S0
    err = tolerance + 1
    i = 1
    while err > tolerance && i <= maxiter
        next_S = A * S + A' * V
        err = norm(S - next_S)
        S = next_S
        i = i + 1
    end

    return S
end
```
\[ A = \begin{bmatrix} 0.8 & -0.2 \\ -0.1 & 0.7 \end{bmatrix} \]
\[ \Sigma = \begin{bmatrix} 0.5 & 0.4 \\ 0.4 & 0.6 \end{bmatrix} \]

Note that all eigenvalues of \( A \) lie inside the unit disc:

\[
\text{maximum(abs, eigvals}(A))
\]

Let's compute the asymptotic variance:

\[
0.9
\]

\[
\text{compute_asymptotic_var}(A, \Sigma)
\]

\[
2\times2 \text{Array}\{\text{Float64,2}:  \\
0.671228 0.633476  \\
0.633476 0.858874  
\}
\]

Now let's do the same thing using QuantEcon's `solve_discrete_lyapunov()` function and check we get the same result:

\[
\text{using QuantEcon}
\]

\[
\text{solve_discrete_lyapunov}(A, \Sigma * \Sigma')
\]

\[
2\times2 \text{Array}\{\text{Float64,2}:  \\
0.671231 0.633474  \\
0.633474 0.858874  
\}
\]

1.5 Types, Methods and Dispatch

Contents

- Types, Methods and Dispatch
  - Overview
  - Types and Multiple Dispatch
  - The Type Hierarchy
  - User-Defined Types
  - Exercises
  - Solutions
1.5.1 Overview

In this lecture we delve more deeply into the structure of Julia, and in particular into
- the concept of types
- methods and multiple dispatch
- building user-defined types

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

Understanding them will help you
- Write well organized Julia code that's easy to read, modify, maintain and debug
- Improve the speed at which your code runs
- Read Julia code written by other programmers

1.5.2 Types and Multiple Dispatch

Common Types

In Julia all objects have a type, which can be queried using the `typeof()` function

```julia
typeof(0.5)
```

Float64

```julia
typeof(5)
```

Int64

```julia
typeof("foo")
```

String

```julia
typeof('c')
```

Char

The next two types use curly bracket notation to express the fact that they are *parametric*

```julia
typeof(1 + im)
```

Complex{Int64}

```julia
typeof(eye(2))
```

Chapter 1. Programming in Julia
Well return to parametric types later in this lecture.

Remark: Note that, by convention, type names use CamelCase: `FloatingPoint`, `Array`, `AbstractArray`, etc.

**Variables and Type**

After assigning a variable name to an object, we can query the type of the object via the name:

```julia
x = 42
```

```julia
42
```

```julia
typeof(x)
```

```julia
Int64
```

The type resides with the object itself, not with the name `x`.

Thus, `x` is just a symbol bound to an object of type `Int64`.

Indeed, we can *rebind* the symbol `x` to any other object, of the same type or otherwise:

```julia
x = 42.0
```

```julia
42.0
```

Now `x` points to another object, of type `Float64`.

```julia
typeof(x)
```

```julia
Float64
```

**Multiple Dispatch**

When we process data with a computer, the precise data type is important—sometimes more than we realize.

For example, on an abstract mathematical level, we don’t distinguish between `1 + 1` and `1.0 + 1.0`.

But for a CPU, integer and floating point addition are different things, using a different set of instructions.

Julia handles this problem by storing multiple, specialized versions of functions like addition, one for each data type or set of data types.

These individual specialized versions are called *methods*.

When an operation like addition is requested, the Julia runtime environment inspects the type of data to be acted on and hands it out to the appropriate method.
This process is called **multiple dispatch**

**Example 1**

In Julia, $1 + 1$ has the alternative syntax $+(1, 1)$

```
 +(1, 1)
```

```
2
```

This operator $+$ is itself a function with multiple methods.

We can investigate them using the `@which` macro, which shows the method to which a given call is dispatched.

```
x, y = 1.0, 1.0
@which +(x, y)
```

```
+(x::Float64, y::Float64) at float.jl:240
```

We see that the operation is sent to the $+$ method that specializes in adding floating point numbers.

Heres the integer case.

```
x, y = 1, 1
@which +(x, y)
```

```
+(T<:Union{Int128, Int32, Int64, ...})(x::T, y::T) at int.jl:32
```

This (slightly edited) output says that the call has been dispatched to the $+$ method responsible for handling integer values.

(Well learn more about the details of this syntax below)

Heres another example, with complex numbers.

```
x, y = 1.0 + 1.0im, 1.0 + 1.0im
@which +(x, y)
```

```
+(z::Complex, w::Complex) at complex.jl:125
```

Again, the call has been dispatched to a $+$ method specifically designed for handling the given data type.

**Example 2**

The `isless` function also has multiple methods.

```
isless(1.0, 2.0)  # Applied to two floats
```
true

@which isless(1.0, 2.0)

isless(x::Float64, y::Float64) at float.jl:283

Now let's try with integers

@which isless(1, 2)

isless(x::Real, y::Real) at operators.jl:75

The `Real` data type we haven't met yet; it's an example of an *abstract* type, and encompasses both floats and integers.

We'll learn more about abstract types below.

**Example 3**

The function `isfinite()` has multiple methods too

@which isfinite(1) # Call isfinite on an integer

isfinite(x::Integer) at float.jl:360

@which isfinite(1.0) # Call isfinite on a float

isfinite(x::AbstractFloat) at float.jl:358

Here `AbstractFloat` is another abstract data type, this time encompassing all floats.

We can list all the methods of `isfinite` as follows

methods(isfinite)

9 methods for generic function isfinite:
isfinite(x::BigFloat) at mpfr.jl:799
isfinite(x::Float16) at float16.jl:119
isfinite(x::AbstractFloat) at float.jl:358
isfinite(x::Integer) at float.jl:360
isfinite(::Irrational) at irrationals.jl:82
isfinite(x::Real) at float.jl:359
isfinite(x::Complex) at complex.jl:57
isfinite{T<:Number}(x::AbstractArray{T,N<:Any}) at operators.jl:555
isfinite{T<:Base.Dates.TimeType}(::Union{T,Type{T}}) at dates/types.jl:218

We'll discuss some of the more complicated data types you see here later on.
Adding Methods

It's straightforward to add methods to existing functions.

For example, we can't at present add an integer and a string in Julia

```julia
+(100, "100")
```

MethodError: no method matching +(::Int64, ::String)

This is sensible behavior, but if you want to change it there's nothing to stop you:

```julia
importall Base.Operators  # Gives access to + so that we can add a method
+(x::Integer, y::String) = x + parse(Int, y)
```

```julia
+(100, "100")
```

200

100 + "100"

200

Dispatch and User-Defined Functions

You can exploit multiple dispatch in user-defined functions.

Here's a trivial example (we'll see many realistic examples later)

```julia
function h(a::Float64)
    println("You have called the method for handling Float64s")
end

function h(a::Int64)
    println("You have called the method for handling Int64s")
end
```

The $h$ that gets invoked depends on the data type that you call it with:

```julia
h(1.0)
```

You have called the method for handling Float64s
You have called the method for handling Int64s

Actually, as well see when we *discuss JIT compilation*, this process is partly automated

For example, if we write a function that can handle either floating point or integer arguments and then call it with floating point arguments, a specialized method for applying our function to floats will be constructed and stored in memory

- Inside the method, operations such as addition, multiplication, etc. will be specialized to their floating point versions

If we next call it with integer arguments, the process will be repeated but now specialized to integers

- Inside the method, operations such as addition, multiplication, etc. will be specialized to their integer versions

Subsequent calls will be routed automatically to the most appropriate method

**Comments on Efficiency**

Julia’s multiple dispatch approach to handling data differs from the approach used by many other languages

It is, however, well thought out and well suited to scientific computing

The reason is that many methods, being specialized to specific data types, are highly optimized for the kind of data that they act on

We can likewise build specialized methods and hence generate fast code

Well see how this enables Julia to easily generate highly efficient machine code in *later on*

### 1.5.3 The Type Hierarchy

Lets discuss how types are organized in Julia

**Abstract vs Concrete Types**

We saw above that `Float64` is the standard type for representing a 64 bit floating point number

But weve also seen references to types such as `Real` and `AbstractFloat`

The former (i.e., `Float64`) is an example of a **concrete type**, as is `Int64` or `Float32`

The latter (i.e., `Real`, `AbstractFloat`) are examples of so-called **abstract types**

Concrete types are types that we can *instantiate* i.e., pair with data in memory

On the other hand, abstract types help us organize and work with related concrete types
The Type Hierarchy

How exactly do abstract types organize or relate different concrete types?

The answer is that, in the Julia language specification, the types form a hierarchy.

For example, Float64 and Int64 are subtypes of Real.

\[
\text{Float64} \subseteq \text{Real} \quad \text{true}
\]

\[
\text{Int64} \subseteq \text{Real} \quad \text{true}
\]

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

\[
\text{Complex64} \not\subseteq \text{Real} \quad \text{false}
\]

They are, however, subtypes of Number.

\[
\text{Complex64} \subseteq \text{Number} \quad \text{true}
\]

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

\[
\text{Number} \subseteq \text{Any} \quad \text{true}
\]

In particular, the type tree is organized with Any at the top and the concrete types at the bottom.

We never actually see instances of abstract types (i.e., typeof(x) never returns an abstract type).

The point of abstract types is to categorize the concrete types, as well as other abstract types that sit below them in the hierarchy.

Back to 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 `AbstractFloat` and `Number` respectively.

```
supertype(Float64)
```

```
AbstractFloat
```

```
supertype(Complex64)
```

```
Number
```

Hence the interpreter looks next for a method of the form $f(x::AbstractFloat, y::Number)$.

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 following error:

```
+(100, "100")
```

```
MethodError: no method matching +(::Int64, ::String)
  ...
```

Because the dispatch procedure starts from concrete types and works upwards, dispatch always invokes the most specific method 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 optimized for 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.

Here’s another simple example, involving a user-defined function:

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

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

function f(x::Integer)
```

1.5. Types, Methods and Dispatch
println("Integer method invoked")

end

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

- \( f(3) \)

  Integer method invoked

- \( f(3.0) \)

  Number method invoked

- \( f("foo") \)

  Generic function invoked

Since

- 3 is an \texttt{Int64} and \texttt{Int64 <: Integer <: Number}

the call \( f(3) \) proceeds up the tree to \texttt{Integer} and invokes \( f(x::\texttt{Integer}) \)

On the other hand, \( 3.0 \) is a \texttt{Float64}, which is not a subtype of \texttt{Integer}

Hence the call \( f(3.0) \) continues up to \( f(x::\texttt{Number}) \)

Finally, \( f("foo") \) is handled by the generic function, since \texttt{String} is not a subtype of \texttt{Number}

### 1.5.4 User-Defined Types

Let's have a look at defining our own data types

**Motivation**

At our respective homes we both have draws full of fishing gear

Of course we have draws full of other things too, like kitchen utensils, or clothes

Are these draws really necessary?

Perhaps not, but who wants to search the whole house for their fishing reel when the fish are biting?

Certainly not us

Just as it's convenient to store household objects in draws, its also convenient to organize the objects in your program into designated containers

The first step is to design and build the containers

We do this by declaring and using our own types
For example,

- a `Firm` type might store parameters for objects that represent firms in a given model
- an `EstimationResults` type might store output from some statistical procedure, etc.

Once those types are declared, we can create instances of the type

For example,

```julia
results = EstimationResults(y, X)
```

might create an instances of `EstimationResults` that stores estimated coefficients and other information from a given regression exercise involving data $y, X$

**Syntax**

While there are multiple ways to create new types, we almost always use the `struct` keyword, which is for creation of composite data types

Notes:

- composite refers to the fact that the data types in question can be used as containers that hold a variety of data
- the `struct` terminology is used in a number of programming languages to refer to composite data types

Let's start with a trivial example where the `struct` we build is empty

```julia
struct Foo  # A useless data type that stores no data
end
```

When a new data type is defined in this way, the interpreter also 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:

```julia
foo = Foo()  # Call default constructor, make a new Foo
```

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

```julia
typeof(foo)

Foo
```

**Adding Methods**

We can now create functions that act on instances of `Foo`

1.5. Types, Methods and Dispatch
foofunc(x::Foo) = "onefoo"

foofunc (generic function with 1 method)

foofunc(foo)

"onefoo"

Or we can add new methods for acting on Foos to existing functions, such as +

+(x::Foo, y::Foo) = "twofoos"

+ (generic function with 165 methods)

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

(Foo(), Foo())

+(foo1, foo2)

"twofoos"

A Less Trivial Example

Let’s say we are doing a lot of work with AR(1) processes, which are random sequences \( \{X_t\} \) that follow the law of motion

\[
X_{t+1} = aX_t + b + \sigma W_{t+1}
\]  \hspace{1cm} (1.5)

Here

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

Let’s take these primitives \( a, b, \sigma \) and \( \phi \) and organize them into a single entity like so

mutable struct AR1
        a
        b
        \sigma
    end

Here mutable means that we can change (mutate) data while the object is live in memory – see below

For the distribution we’ll assign a Distribution from the Distributions package
```julia
using Distributions

m = AR1(0.9, 1, 1, Beta(5, 5))

AR1(0.9,1,1,Distributions.Beta{Float64}(α=5.0, β=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:

- `m.a` 0.9
- `m.b` 1
- `m.σ` 1
- `m` Distributions.Beta{Float64}(α=5.0, β=5.0)

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

```julia
typeof(m.)
```

Distributions.Beta{Float64}

```julia
typeof(m.) <: Distribution
```

true

We can reach into m and change this if we want to:

```julia
m. = Exponential(0.5)
```

Distributions.Exponential{Float64}(θ=0.5)
Specifying Field Types

In our type definition we can be explicit that we want to be a `Distribution` and the other elements to be floats

```julia
struct AR1_explicit
    a::Float64
    b::Float64
    σ::Float64
    ::Distribution
end
```

(In this case, mutable is removed since we do not intend to make any changes to the elements of `AR1_explicit`)

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

```julia
m = AR1_explicit(0.9, 1, "foo", Beta(5, 5))
```

MethodError: Cannot `convert` an object of type String to an object of type Real

This can be useful in terms of failing early on incorrect data, rather than deeper into execution

At the same time, `AR1_explicit` is not as generic as `AR1`, and hence less flexible

For example, suppose that we want to allow `a`, `b` and `σ` to take any value that is `<: Real`

We could achieve this by the new definition

```julia
struct AR1_real
    a::Real
    b::Real
    σ::Real
    ::Distribution
end
```

But it turns out that using abstract types inside user-defined types adversely affects performance. More about that soon

Fortunately, there is another approach that both

- preserves the use of concrete types for internal data and
- allows flexibility across multiple concrete data types

This approach uses type parameters, a topic we turn to now

Type Parameters

Consider the following output
Here \texttt{Array} is one of Julia's predefined types (Array <: DenseArray <: AbstractArray <: Any).

The \texttt{Int64,1} in curly brackets are \textbf{type parameters}.

In this case they are the element type and the dimension.

Many other types have type parameters too.

\begin{verbatim}
typeof(1.0 + 1.0im)

type(T::Complex{Float64})

type(T::Complex{Int64})
\end{verbatim}

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

We can use parametric types in our own type definitions, as the next example shows.

\begin{verbatim}
Back to the AR1 Example

Recall our AR(1) example, where we considered different restrictions on internal data.

For the coefficients $a$, $b$ and $\sigma$ we considered:

- allowing them to be any type
- forcing them to be of type \texttt{Float64}
- allowing them to be any \texttt{Real}

The last option is a nice balance between specific and flexible.

For example, using \texttt{Real} in the type definition tells us that, while these values should be scalars, integer values and floats are both OK.

However, as mentioned above, using abstract types for fields of user-defined types impacts negatively on performance.

For now it suffices to observe that we can achieve flexibility and eliminate abstract types on $a$, $b$ and $\sigma$ by the following declaration:

\begin{verbatim}
struct AR1_best{T <: Real}
  a::T
  b::T
  \sigma::T
\end{verbatim}

1.5. Types, Methods and Dispatch
If we create an instance using `Float64` values then the instance has type `AR1_best{Float64}`

\[
m = AR1\_best(0.9, 1.0, 1.0, \text{Beta}(5, 5))
\]

**1.5.5 Exercises**

**Exercise 1**

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

- an instance \( m \) of \( \text{AR1} \) (see above)
- 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 \( \text{AR}(1) \) process are as specified in \( m \)
- the initial condition \( X_0 \) is set equal to \( x_0 \)

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

**1.5.6 Solutions**

**Exercise 1**

Let's start with the \( \text{AR1} \) definition as specified in the lecture

```julia
struct AR1_ex1{T <: Real}
    a::T
    b::T
    σ::T
    ::Distribution
end
```

Now let's write the function to simulate \( \text{AR1s} \)

```julia
function simulate(m::AR1_ex1, n::Integer, x0::Real)
    X = Array{Float64}(n)
    X[1] = x0
    for t in 1:(n-1)
        # Implementation
    end
end
```
\[ X[t+1] = m.a \times X[t] + m.b + m.\sigma \times \text{rand}(m.) \]

```
end
return X
end
```

Let's test it out on the AR(1) process discussed in the lecture

```julia
m = AR1_ex1(0.9, 1.0, 1.0, Beta(5, 5))
X = simulate(m, 100, 0.0)
```

```
100-element Array{Float64,1}:
  0.0
  1.43703
  2.75257
  4.07441
  5.08574
  6.24482
  7.19393
  7.9938
  8.49046
  9.12257
  9.77304
 10.0554
 10.3597
 14.8071
 14.4791
 14.3067
 14.4508
 14.7374
 14.854
 14.7483
 14.8965
 14.6211
 14.7057
 14.6627
 14.7309
```

Next let's plot the time series to see what it looks like

```julia
using Plots
plot(X, legend=:none)
```
1.6 The Need for Speed

Contents

- The Need for Speed
  - Overview
  - Foundations
  - JIT Compilation in Julia
  - Fast and Slow Julia Code
  - Further Comments

1.6.1 Overview

Computer scientists often classify programming languages according to the following two categories

*High level languages* aim to maximize productivity by

- being easy to read, write and debug
- automating standard tasks (e.g., memory management)
• being interactive, etc.

Low level languages aim for speed and control, which they achieve by

• being closer to the metal (direct access to CPU, memory, etc.)
• requiring a relatively large amount of information from the user (e.g., all data types must be specified)

Traditionally we understand this as a trade off

• high productivity or high performance
• optimized for humans or optimized for machines

One of the great strengths of Julia is that it pushes out the curve, achieving both high productivity and high performance with relatively little fuss.

The word relatively is important here, however

In simple programs, excellent performance is often trivial to achieve

For longer, more sophisticated programs, you need to be aware of potential stumbling blocks

This lecture covers the key points

Requirements

You should read our earlier lecture on types, methods and multiple dispatch before this one

1.6.2 Foundations

Let's think about how quickly code runs, taking as given

• hardware configuration
• algorithm (i.e., set of instructions to be executed)

Well start by discussing the kinds of instructions that machines understand

Machine Code

All instructions for computers end up as machine code

Writing fast code expressing a given algorithm so that it runs quickly boils down to producing efficient machine code

You can do this yourself, by hand, if you want to

Typically this is done by writing assembly, which is a symbolic representation of machine code

Here's some assembly code implementing a function that takes arguments $a, b$ and returns $2a + 8b$
Note that this code is specific to one particular piece of hardware that we use. Different machines require different machine code.

If you ever feel tempted to start rewriting your economic model in assembly, please restrain yourself. It’s far more sensible to give these instructions in a language like Julia, where they can be easily written and understood.

In any of these languages we end up with code that is much easier for humans to write, read, share and debug. We leave it up to the machine itself to turn our code into machine code. How exactly does this happen?

**Generating Machine Code**

The process for turning high level code into machine code differs across languages. Let’s look at some of the options and how they differ from one another.

**AOT Compiled Languages**

Traditional compiled languages like Fortran, C and C++ are a reasonable option for writing fast code.
Indeed, the standard benchmark for performance is still well-written C or Fortran. These languages compile down to efficient machine code because users are forced to provide a lot of detail on data types and how the code will execute. The compiler therefore has ample information for building the corresponding machine code ahead of time (AOT) in a way that

- organizes the data optimally in memory and
- implements efficient operations as required for the task in hand

At the same time, the syntax and semantics of C and Fortran are verbose and unwieldy when compared to something like Julia. Moreover, these low level languages lack the interactivity that's so crucial for scientific work.

**Interpreted Languages**

Interpreted languages like Python generate machine code on the fly, during program execution. This allows them to be flexible and interactive. Moreover, programmers can leave many tedious details to the runtime environment, such as

- specifying variable types
- memory allocation/deallocation, etc.

But all this convenience and flexibility comes at a cost: it's hard to turn instructions written in these languages into efficient machine code.

For example, consider what happens when Python adds a long list of numbers together. Typically, the runtime environment has to check the type of these objects one by one before it figures out how to add them. This involves substantial overheads.

There are also significant overheads associated with accessing the data values themselves, which might not be stored contiguously in memory.

The resulting machine code is often complex and slow.

**Just-in-time compilation**

Just-in-time (JIT) compilation is an alternative approach that marries some of the advantages of AOT compilation and interpreted languages. The basic idea is that functions for specific tasks are compiled as requested. As long as the compiler has enough information about what the function does, it can in principle generate efficient machine code. In some instances, all the information is supplied by the programmer.
In other cases, the compiler will attempt to infer missing information on the fly based on usage. Through this approach, computing environments built around JIT compilers aim to:

- provide all the benefits of high level languages discussed above and, at the same time,
- produce efficient instruction sets when functions are compiled down to machine code.

### 1.6.3 JIT Compilation in Julia

JIT compilation is the approach used by Julia. In an ideal setting, all information necessary to generate efficient native machine code is supplied or inferred. In such a setting, Julia will be on par with machine code from low level languages.

#### An Example

Consider the function:

```julia
function f(a, b)
    y = (a + 8b)^2
    return 7y
end
```

Suppose we call `f` with integer arguments (e.g., `z = f(1, 2)`). The JIT compiler now knows the types of `a` and `b`. Moreover, it can infer types for other variables inside the function:

- e.g., `y` will also be an integer.

It then compiles a specialized version of the function to handle integers and stores it in memory. We can view the corresponding machine code using the `@code_native` macro:

```julia
@code_native f(1, 2)
```

```assembly
pushq  %rbp
movq  %rsp, %rbp
leaq  (%rdi,%rsi,8), %rdi
movabsq $power_by_squaring, %rax
movl  $2, %esi
callq %rax
imulq $7, %rax, %rax
popq  %rbp
retq
nop
```

If we now call `f` again, but this time with floating point arguments, the JIT compiler will once more infer types for the other variables inside the function:

- e.g., `y` will also be a float.
It then compiles a new version to handle this type of argument

```julia
@code_native f(1.0, 2.0)
```

```
pushq %rbp
movq %rsp, %rbp
movabsq $139613711993752, %rax # imm = 0x7EFA59B58B98
mulsd (%rax), %xmm1
addsd %xmm0, %xmm1
mulsd %xmm1, %xmm1
movabsq $139613711993760, %rax # imm = 0x7EFA59B58BA0
mulsd (%rax), %xmm1
movapd %xmm1, %xmm0
popq %rbp
retq
nop
```

Subsequent calls using either floats or integers are now routed to the appropriate compiled code.

**Potential Problems**

In some senses, what we saw above was a best case scenario. Sometimes the JIT compiler produces messy, slow machine code. This happens when type inference fails or the compiler has insufficient information to optimize effectively. The next section looks at situations where these problems arise and how to get around them.

**1.6.4 Fast and Slow Julia Code**

To summarize what we’ve learned so far, Julia provides a platform for generating highly efficient machine code with relatively little effort by combining

1. JIT compilation
2. Optional type declarations and type inference to pin down the types of variables and hence compile efficient code
3. Multiple dispatch to facilitate specialization and optimization of compiled code for different data types

But the process is not flawless, and hiccups can occur. The purpose of this section is to highlight potential issues and show you how to circumvent them.

**Global Variables**

Global variables are names assigned to values outside of any function or type definition. They are convenient and novice programmers typically use them with abandon. But global variables are also dangerous, especially in medium to large size programs, since...
• they can affect what happens in any part of your program
• they can be changed by any function

This makes it much harder to be certain about what some small part of a given piece of code actually commands

Here’s a useful discussion on the topic

When it comes to JIT compilation, global variables create further problems

The reason is that the compiler can never be sure of the type of the global variable, or even that the type will stay constant while a given function runs

To illustrate, consider this code, where \( b \) is global

```julia
b = 1.0
function g(a)
    for i in 1:1_000_000
        tmp = a + b
    end
end
```

The code executes relatively slowly and uses a huge amount of memory

```julia
@time g(1.0)
```

```
0.023039 seconds (2.00 M allocations: 30.518 MB, 12.90% gc time)
```

If you look at the corresponding machine code you will see that its a mess

```julia
@code_native g(1.0)
```

```assembly
pushq %rbp
movq %rsp, %rbp
pushq %r15
pushq %r14
pushq %r13
pushq %r12
pushq %rbx
subq $56, %rsp
movsd %xmm0, -88(%rbp)
movq %fs:0, %r15
addq $-2672, %r15 # imm = 0xFFFFFFFFFFFFF590
xorpd %xmm0, %xmm0
movupd %xmm0, -64(%rbp)
movq $0, -48(%rbp)
movq $6, -80(%rbp)
movq (%r15), %rax
leaq -80(%rbp), %rax
movabsq $140242067578672, %r12 # imm = 0x7F8CA69EDF30
movl $1000000, %ebx # imm = 0xF4240
```
leaq  5596992(%r12), %r13  
movabsq $jl_apply_generic, %r14  
nop
movq  75966904(%r12), %rax  
movq  %rax, -48(%rbp)  
movq  %r13, -64(%rbp)  
movl  $1432, %esi  
        # imm = 0x598  
movl  $16, %edx
movq  %r15, %rdi
movabsq $jl_gc_pool_alloc, %rax  
callq %rax
movq  %r12, -8(%rax)  
movsd -88(%rbp), %xmm0  
        # xmm0 = mem[0], zero
movsd %xmm0, (%rax)  
movq  %rax, -56(%rbp)
leaq -64(%rbp), %rdi
callq %r14
decq %rbx
jne L112
movq -72(%rbp), %rax
movq  %rax, (%r15)
addq $56, %rsp
popq  %rbx
popq  %r12
popq  %r13
popq  %r14
popq  %r15
popq  %r14
retq
nopw  %cs:(%rax,%rax)

If we eliminate the global variable like so

```julia
function g(a, b)
    for i in 1:1_000_000
        tmp = a + b
    end
end
```

then execution speed improves dramatically

```julia
@time g(1.0, 1.0)
```

0.002876 seconds (1.31 k allocations: 61.374 KB)

```julia
@time g(1.0, 1.0)
```

0.000001 seconds (4 allocations: 160 bytes)

Note that the second run was dramatically faster than the first

1.6. The Need for Speed
Thats because the first call included the time for JIT compilation

Notice also how small the memory footprint of the execution is

Also, the machine code is simple and clean

```julia
@code_native g(1.0, 1.0)

pushq %rbp
movq %rsp, %rbp
popq %rbp
retq
nopw %cs:(%rax,%rax)
```

Now the compiler is certain of types throughout execution of the function and hence can optimize accordingly

**The const keyword**

Another way to stabilize the code above is to maintain the global variable but prepend it with `const`

```julia
const b_const = 1.0
function g(a)
    for i in 1:1_000_000
        tmp = a + b_const
    end
end
```

Now the compiler can again generate efficient machine code

Well leave you to experiment with it

**Composite Types with Abstract Field Types**

Another scenario that trips up the JIT compiler is when composite types have fields with abstract types

We met this issue *earlier*, when we discussed AR(1) models

Lets experiment, using, respectively,

- an untyped field
- a field with abstract type, and
- parametric typing

As well see, the last of options these gives us the best performance, while still maintaining significant flexibility

Heres the untyped case
struct Foo_generic
    a
end

Here's the case of an abstract type on the field a

struct Foo_abstract
    a::Real
end

Finally, here's the parametrically typed case

struct Foo_concrete{T <: Real}
    a::T
end

Now we generate instances

fg = Foo_generic(1.0)
fa = Foo_abstract(1.0)
fcc = Foo_concrete(1.0)

In the last case, concrete type information for the fields is embedded in the object

typeof(fc)

Foo_concrete{Float64}

This is significant because such information is detected by the compiler

**Timing**

Here's a function that uses the field a of our objects

```julia
function f(foo)
    for i in 1:1_000_000
        tmp = i + foo.a
    end
end
```

Lets try timing our code, starting with the generic case:

```julia
@time f(fg)
```

0.029499 seconds (2.00 M allocations: 30.510 MB, 21.81% gc time)

The timing is not very impressive

Here's the nasty looking machine code
The abstract case is similar
@time f(fa)

0.030892 seconds (2.00 M allocations: 30.585 MB, 7.18% gc time)

Note the large memory footprint.

The machine code is also long and complex, although we omit details.

Finally, let's look at the parametrically typed version.

@time f(fc)

0.002850 seconds (1.45 k allocations: 67.642 KB)

Some of this time is JIT compilation, and one more execution gets us down to

0.000001 seconds (3 allocations: 144 bytes)

Here's the corresponding machine code.

@code_native f(fc)

```
pushq %rbp
movq %rsp, %rbp
popq %rbp
retq
nopw %cs: (%rax, %rax)
```

Much nicer.

**Abstract Containers**

Another way we can run into trouble is with abstract container types.

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.

```
x = linspace(0, 1, 1e6)
x = collect(x)
typeof(x)
```
Array{Float64,1}

@time sum_float_array(x)

0.005524 seconds (1.74 k allocations: 82.486 KB)

When Julia compiles this function, it knows that the data passed in as x will be an array of 64 bit floats
Hence its 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

Heres the same function minus the type annotation in the function signature

```julia
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 at a similar speed as the function
with type information

```
@time sum_array(x)
```

0.005556 seconds (1.61 k allocations: 75.002 KB)

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 produced to handle that type
In this case, since were 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()

An Abstract Container

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
x = Any[1/i for i in 1:1e6];
```

Chapter 1. Programming in Julia
eltype(x)

Any

Now summation is much slower and memory management is less efficient

@time sum_array(x)

0.021680 seconds (1.00 M allocations: 15.332 MB)

1.6.5 Further Comments

Here are some final comments on performance

Explicit Typing

Writing fast Julia code amounts to writing Julia from which the compiler can generate efficient machine code

For this, Julia needs to know about the type of data its processing as early as possible

We could hard code the type of all variables and function arguments but this comes at a cost

Our code becomes more cumbersome and less generic

We are starting to lose the advantages that drew us to Julia in the first place

Moreover, explicitly typing everything is not necessary for optimal performance

The Julia compiler is smart and can often infer types perfectly well, without any performance cost

What we really want to do is

- keep our code simple, elegant and generic
- help the compiler out in situations where its liable to get tripped up

Summary and Tips

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

A good next stop for further reading is the relevant part of the Julia documentation

1.6. The Need for Speed 109
1.7 Plotting in Julia

1.7.1 Overview

Since its inception, plotting in Julia has been a mix of happiness and frustration. Some initially promising libraries have stagnated, or failed to keep up with user needs. New packages have appeared to compete with them, but not all are fully featured. The good news is that the Julia community now has several very good options for plotting.

In this lecture, we try to save you some of our pain by focusing on what we believe are currently the best libraries.

First, we look at two high-quality plotting packages that have proved useful to us in a range of applications. After that, we turn to a relative newcomer called Plots.jl.

The latter package takes a different—and intriguing—approach that combines and exploits the strengths of several existing plotting libraries.

Below we assume that:

- you’ve already read through our getting started lecture
- you are working in a Jupyter notebook, as described here

How to Read this Lecture

If you want to get started quickly with relatively simple plots, you can skip straight to the section on Plots.jl.

If you want a deeper understanding and more flexibility, continue from the next section and read on.

Credits: Thanks to @albep, @vgregory757, and @spencerlyon2 for help with the code examples below.
1.7.2 PyPlot

Lets look at PyPlot first

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

Installing PyPlot

One disadvantage of PyPlot is that it not only requires Python but also much of the scientific Python stack

Fortunately, installation of the latter has been greatly simplified by the excellent Anaconda Python distribution

Moreover, the tools that come with Anaconda (such as Jupyter) are too good to miss out on

So please go ahead and install Anaconda if you havent yet

Next, start up Julia and type Pkg.add("PyPlot")

Usage

There are two different interfaces to Matplotlib and hence to PyPlot

Lets look at them in turn

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

Heres an example

```julia
using PyPlot
x = linspace(0, 10, 200)
y = sin.(x)
plot(x, y, "b-", linewidth=2)
```
3D Plots

Here’s an example of how to create a 3D plot

```julia
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

xgrid, ygrid = meshgrid(x, y)
surf(xgrid, ygrid, z', cmap=ColorMap("jet"), alpha=0.7)
zlim(-0.5, 1.0)

It creates this figure
```
The Object Oriented API

Matplotlib also has a more powerful and expressive object oriented API.

Because Julia isn't object oriented in the same sense as Python, the syntax required to access this interface via PyPlot is a little awkward.

Here's an example:

```julia
x = linspace(0, 10, 200)
y = sin.(x)
fig, ax = subplots()
ax[:plot](x, y, "b-", linewidth=2)
```
The resulting figure is the same

Here we get no particular benefit from switching APIs, while introducing a less attractive syntax

However, as plots get more complex, the more explicit syntax will give us greater control

Heres a similar plot with a bit more customization

```julia
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

```julia
x = linspace(0, 10, 200)
y = sin.(x)
fig, ax = subplots()
ax[:, plot](x, y, "r-", linewidth=2, label=L"$y = \sin(x)$", alpha=0.6)
ax[:, legend](loc="upper center")
```
Multiple Plots on One Axis

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

We use `Distributions.jl` to get the values of the densities given a randomly generated mean and standard deviation.

```julia
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]()
```

Note the `\` in front of the string to indicate LaTeX markup.
A figure containing \( n \) rows and \( m \) columns of subplots can be created by the call

\[
\text{fig, axes} = \text{subplots}(\text{num\_rows}, \text{num\_cols})
\]

Here is an example that generates 6 normal distributions, takes 100 draws from each, and plots each of the resulting histograms:

```julia
u = Uniform()
num_rows, num_cols = 2, 3
fig, axes = subplots(num_rows, num_cols, figsize=(16,6))
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 ==#
```
1.7.3 PlotlyJS

Now let’s turn to another plotting package—a promising new library called PlotlyJS, authored by Spencer Lyon. PlotlyJS is a Julia interface to the plotly.js visualization library. It can be installed by typing `Pkg.add("PlotlyJS")` from within Julia. It has several advantages, one of which is beautiful interactive plots. While we won’t treat the interface in great detail, we will frequently use PlotlyJS as a backend for Plots.jl (more on this below).

Examples

Let’s look at some simple examples. Here’s a version of the sine function plot you saw above:

```julia
import Plots
x = linspace(0, 10, 200)
y = sin.(x)
# specify which module scatter belongs to since both have scatter
Plots.plot(Plots.scatter(x=x, y=y, marker_color="blue", line_width=2))
```
Here's a replication of the figure with multiple Gaussian densities.

```julia
traces = PlotlyJS.GenericTrace[]
u = Uniform()

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
dx = Normal(m, s)
y = pdf.(d, x)
    trace = PlotlyJS.scatter(x=x, y=y, name="draw $i")
push!(traces, trace)
end
PlotlyJS.plot(traces, PlotlyJS.Layout())
```

The output looks like this (modulo randomness):
1.7.4 Plots.jl

Plots.jl is another relative newcomer to the Julia plotting scene, authored by Tom Breloff.

The approach of Plots.jl is to

1. provide a frontend plotting language
2. render the plots by using one of several existing plotting libraries as backends

In other words, Plots.jl plotting commands are translated internally to commands understood by a selected plotting library.

Underlying libraries, or backends, can be swapped very easily.

This is neat because each backend has a different look, as well as different capabilities.

Also, Julia being Julia, its quite possible that a given backend wont install or function on your machine at a given point in time.

With Plots.jl, you can just change to another one.

Simple Examples

We produced some simple plots using Plots.jl back in our introductory Julia lecture.

Here's another simple one:

```julia
import Plots
x = linspace(0, 10, 200)
y = sin.(x)
Plots.plot(x, y, color=:blue, linewidth=2, label="sine")
```

No backend was specified in the preceding code, and in this case it defaulted to Plots.jl.
We can make this explicit by adding one extra line

```julia
Plots.pyplot()  # specify backend
x = linspace(0, 10, 200)
y = sin.(x)
Plots.plot(x, y, color=:blue, linewidth=2, label="sine")
```

To switch your backend to PlotlyJS, change `pyplot()` to `plotlyjs()`

Your figure should now look more like the plots produced by PlotlyJS

Here’s a slightly more complex plot using Plots.jl with PyPlot backend

```julia
using LaTeXStrings  # Install this package
Plots.pyplot()
x = linspace(0, 10, 100)
Plots.plot(x, sin,
    color=:red,
    lw=2,
    yticks=-1:1:1,
    title="sine function",
    label=L"y = \sin(x)",  # L for LaTeX string
    alpha=0.6)
```

Use `legend=:none` if you want no legend on the plot

Notice that in the preceding code example, the second argument to `plot()` is a function rather than an array of data points

1.7. Plotting in Julia
This is valid syntax, as is

```
Plots.plot(sin, 0, 10)  # Plot the sine function from 0 to 10
```

Plots.jl accommodates these useful variations in syntax by exploiting multiple dispatch

**Multiple Plots on One Axis**

Next, let’s replicate the *figure with multiple Gaussian densities*

```
Plots.plotlyjs()

x = linspace(-4, 4, 150)
y_vals = Array{Vector}(3)
labels = Array{String}(1, 3)
for i = 1:3
    m, s = 2*(rand() - 0.5), rand() + 1
    d = Normal(m, s)
    y_vals[i] = pdf.(d, x)
    labels[i] = string("mu = ", round(m, 2))
end
Plots.plot(x, y_vals, linewidth=2, alpha=0.6, label=labels)
```

Also, when you have multiple y-series, *Plots.jl* can accept one x-values vector and apply it to each y-series
Subplots

Let's replicate the subplots figure *shown above*

```julia
Plots.pyplot()

draws = Array{Vector}(6)
titles = Array{String}(1, 6)
for i = 1:6
    m, s = 2*(rand() - 0.5), rand() + 1
    d = Normal(m, s)
    draws[i] = rand(d, 100)
    t = string(L"$\mu = \$", round(m, 2), L", $\sigma = \$", round(s, 2))
    titles[i] = t
end

Plots.histogram(draws,
    layout=6,
    title=titles,
    legend=:none,
    titlefont=Plots.font(9),
    bins=20)
```

Notice that the font and bins settings get applied to each subplot
When you want to pass individual arguments to subplots, you can use a row vector of arguments

- For example, in the preceding code, `titles` is a $1 \times 6$ row vector

Here’s another example of this, with a row vector of different colors for the histograms

```julia
Plots.pyplot()

draws = Array{Vector}(6)
titles = Array{String}(1, 6)
for i = 1:6
    m, s = 2*(rand() - 0.5), rand() + 1
    d = Normal(m, s)
draws[i] = rand(d, 100)
t = string(L"\mu \approx \mu = ", round(m, 2), L", \sigma \approx \sigma = ", round(s, 2))
titles[i] = t
end

Plots.histogram(draws,
    layout=6,
    title=titles,
    legend=:none,
    titlefont=Plots.font(9),
    color=[:red :blue :yellow :green :black :purple],
    bins=20)
```

The result is a bit garish but hopefully the message is clear
3D Plots

Here’s a sample 3D plot

```julia
Plots.plotlyjs()

n = 50
x = linspace(-3, 3, n)
y = x

z = Array{Float64}(n, n)
ff(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] = ff(x[i], y[j])
    end
end
Plots.surface(x, y, z)
```
Further Reading

Hopefully this tutorial has given you some ideas on how to get started with Plots.jl. Well see more examples of this package in action through the lectures. Additional information can be found in the official documentation.

1.7.5 Exercises

Exercise 1

The identity function $f(x) = x$ is approximated on the nonnegative numbers $[0, \infty)$ with increasing degrees of precision by the sequence of functions

$$d_n(x) = \sum_{k=1}^{n2^n} \frac{k-1}{2^n} \mathbb{1}\left\{ \frac{k-1}{2^n} \leq x < \frac{k}{2^n} \right\} + n\mathbb{1}\{x \geq n\}$$

for $n = 1, 2, \ldots$

Here $\mathbb{1}\{P\} = 1$ if the statement $P$ is true and 0 otherwise

(This result is often used in measure theory)
Plot the functions $d_n$, $n = 1, \ldots, 6$ on the interval $[0, 10]$ and compare them to the identity function. Do they get closer to the identity as $n$ gets larger?

### 1.7.6 Solutions

Our aim is to plot the sequence of functions described in the exercise. We will use the library `Plots.jl`.

```julia
Plots.pyplot()

Plots.PyPlotBackend()

Heres the function $d_n$ for any given $n$:

```julia
function d(x, n)
    current_val = 0
    for k in 1:(n * 2^n)
        if (k - 1) / 2^n <= x < k / 2^n
            current_val += (k - 1) / 2^n
        end
    end
    if x >= n
        current_val += n
    end
    return current_val
end
```

```julia
x_grid = linspace(0, 10, 100)
n_vals = [1, 2, 3, 4, 5]

function_vals = []
labels = []

for n in n_vals
    push!(function_vals, [d(x, n) for x in x_grid])
    push!(labels, "$n")
end

push!(function_vals, x_grid)
push!(labels, "identity function")

Plots.plot(x_grid, function_vals,
    label=reshape(labels, 1, length(n_vals) + 1),
    ylim=(0, 10))
```

1.7. Plotting in Julia 127
1.8 Useful Libraries

Contents

- Useful Libraries
  - Overview
  - Distributions
  - Working with Data
  - Interpolation
  - Optimization, Roots and Fixed Points
  - Other Topics
  - Further Reading

1.8.1 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

Credits: Thanks to @cc7768, @vgregory757 and @spencerlyon2 for keeping us up to date with current best practice

### 1.8.2 Distributions

Functions for manipulating probability distributions and generating random variables are supplied by the excellent Distributions.jl package

Well restrict ourselves to a few simple examples (the package itself has detailed documentation)

- \(d = \text{Normal}(m, s)\) creates a normal distribution with mean \(m\) and standard deviation \(s\)
  - defaults are \(m = 0\) and \(s = 1\)
- \(d = \text{Uniform}(a, b)\) creates a uniform distribution on interval \([a, b]\)
  - defaults are \(a = 0\) and \(b = 1\)
- \(d = \text{Binomial}(n, p)\) creates a binomial over \(n\) trials with success probability \(p\)
  - defaults are \(n = 1\) and \(p = 0.5\)

Distributions.jl defines various methods for acting on these instances in order to obtain

- random draws
- evaluations of pdfs (densities), cdfs (distribution functions), quantiles, etc.
- mean, variance, kurtosis, etc.

For example,

- To generate \(k\) draws from the instance \(d\) use \(\text{rand}(d, k)\)
- To obtain the mean of the distribution use \(\text{mean}(d)\)
- To evaluate the probability density function of \(d\) at \(x\) use \(\text{pdf}(d, x)\)

Further details on the interface can be found here

Several multivariate distributions are also implemented
1.8.3 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

commodities = ["crude", "gas", "gold", "silver"]

4-element Array{String,1}:
"crude"
"gas"
"gold"
"silver"

last_price = @data([4.2, 11.3, 12.1, NA])  # Create DataArray

4-element DataArrays.DataArray{Float64,1}:
4.2
11.3
12.1
NA

df = DataFrame(commod=commodities, price=last_price)

4x2 DataFrames.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

df[:,price]
```
The DataFrames package provides a number of methods for acting on DataFrames

A simple one is `describe()`

```julia
describe(df)
```

<table>
<thead>
<tr>
<th>commod</th>
<th>Length</th>
<th>Type</th>
<th>NAs</th>
<th>NA%</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>String</td>
<td>0</td>
<td>0.0%</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>price</th>
<th>Min</th>
<th>1st Qu.</th>
<th>Median</th>
<th>Mean</th>
<th>3rd Qu.</th>
<th>Max</th>
<th>NAs</th>
<th>NA%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.2</td>
<td>7.75</td>
<td>11.3</td>
<td>9.2</td>
<td>11.7</td>
<td>12.1</td>
<td>1</td>
<td>25.0</td>
</tr>
</tbody>
</table>

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](http://juliastats.org) and [JuliaQuant](http://juliaquant.org)

### 1.8.4 Interpolation

In economics we often wish to interpolate discrete data (i.e., build continuous functions that join discrete sequences of points)

We also need such representations to be fast and efficient

The package we usually turn to for this purpose is [Interpolations.jl](http://juliastats.org/Interpolations.jl)
One downside of Interpolations.jl is that the code to set up simple interpolation objects is relatively verbose. The upside is that the routines have excellent performance.

The package is also well written and well maintained.

Another alternative, if using univariate linear interpolation, is LinInterp from QuantEcon.jl.

As we show below, the syntax for this function is much simpler.

**Univariate Interpolation**

Let's start with the univariate case.

We begin by creating some data points, using a sine function.

```julia
using Interpolations
using Plots
plotlyjs()

x = -7:7  # x points, coarse grid
y = sin.(x)  # corresponding y points

xf = -7:0.1:7  # fine grid
plot(xf, sin.(xf), label="sine function")
scatter!(x, y, label="sampled data", markersize=4)
```

Here's the resulting figure.
We can implement linear interpolation easily using QuantEcon’s `LinInterp`

```julia
using QuantEcon

li = LinInterp(x, y)  # create LinInterp object
li(0.3)               # evaluate at a single point
y_linear_qe = li(xf)  # evaluate at multiple points

plot(xf, y_linear_qe, label="linear")
scatter!(x, y, label="sampled data", markersize=4)
```

Here’s the result
The syntax is simple and the code is efficient, but for other forms of interpolation we need a more sophisticated set of routines.

As an example, let’s employ Interpolations.jl to interpolate the sampled data points using piecewise constant, piecewise linear and cubic interpolation.

```julia
itp_const = scale(interpolate(y, BSpline(Constant()), OnGrid()), x)
itp_linear = scale(interpolate(y, BSpline(Linear()), OnGrid()), x)
itp_cubic = scale(interpolate(y, BSpline(Cubic(Line())), OnGrid()), x)
```

When we want to evaluate them at points in their domain (i.e., between min(x) and max(x)) we can do so as follows.

```julia
itp_cubic[0.3]
```

0.29400097760820687

Note the use of square brackets, rather than parentheses!

Let’s plot these objects created above.

```julia
xf = -7:0.1:7
y_const = [itp_const[x] for x in xf]
y_linear = [itp_linear[x] for x in xf]
y_cubic = [itp_cubic[x] for x in xf]
```
Heres the figure we obtain

![Figure showing plots of constant, linear, and cubic interpolations.]

Univariate with Irregular Grid

The `LinInterp` from QuantEcon.jl works the same whether or not the grid is regular (i.e., evenly spaced).

The `Interpolations.jl` code is a bit more picky.

Heres an example of the latter with an irregular grid

```julia
x = log.(linspace(1, exp(4), 10)) + 1  # Uneven grid
y = log.(x)                              # Corresponding y points

itp_const = interpolate((x, ), y, Gridded(Constant()))
itp_linear = interpolate((x, ), y, Gridded(Linear()))

xf = log.(linspace(1, exp(4), 100)) + 1
y_const  = [itp_const[x] for x in xf]
y_linear = [itp_linear[x] for x in xf]
y_true   = [log(x) for x in xf]
```
labels = ["piecewise constant", "linear", "true function"]
plot(xf, [y_const y_linear y_true], label=labels)
scatter!(x, y, label="sampled data", markersize=4, size=(800, 400))

The figure looks as follows

### Multivariate Interpolation

We can also interpolate in higher dimensions

The following example gives one illustration

```julia
n = 5
x = linspace(-3, 3, n)
y = copy(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

itp = interpolate((x, y), z, Gridded(Linear()));

nf = 50
xf = linspace(-3, 3, nf)
yf = copy(xf)

zf = Array{Float64}(nf, nf)
```
This code produces the following figure:

![3D plot](image)

The original function is in blue, while the linear interpolant is shown in green.

### 1.8.5 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

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

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

newton(f, 0.2)
```
The Newton-Raphson method uses local slope information, which can lead to failure of convergence for some initial conditions.

```julia
newton(f, 0.7)
```

\[-1.0022469256696986\]

For this reason most modern solvers use more robust hybrid methods, as does Rootss `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 Brents method.

```julia
using Optim
optimize(x → x^2, -1.0, 1.0)
```

Results of Optimization Algorithm
* Algorithm: Brent's Method
* Search Interval: \([-1.000000, 1.000000]\]
* Minimizer: \(-2.775558\text{e-17}\)
* Minimum: \(7.703720\text{e-34}\)
* Iterations: 5
* Convergence: max(\(|x - x_{\text{upper}}|, |x - x_{\text{lower}}|\)) <= 2*(1.5e-08*|x|+2.2e-16):
  \(\text{true}\)
* 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.

### 1.8.6 Other Topics

### Numerical Integration

The QuadGK library contains a function called `quadgk()` that performs Gaussian quadrature.

```julia
import QuadGK.quadgk
quadgk(x → cos(x), -2pi, 2pi)
```
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
nodes, weights = qnwlege(65, -2*pi, 2*pi);
integral = do_quad(x -> cos.(x), nodes, weights)
```

This gives an integral of `-3.0062757838678067e-15`.

Let's time the two implementations:

```julia
@time quadgk(x -> cos.(x), -2pi, 2pi)
@time do_quad(x -> cos.(x), nodes, weights)
```

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.

**1.8.7 Further Reading**

The full set of libraries available under the Julia packaging system can be browsed at [pkg.julialang.org](http://pkg.julialang.org).
This section of the course contains foundational mathematical and statistical tools and techniques.

## 2.1 Linear Algebra

### Contents

- Linear Algebra
  - Overview
  - Vectors
  - Matrices
  - Solving Systems of Equations
  - Eigenvalues and Eigenvectors
  - Further Topics
  - Exercises
  - Solutions

### 2.1.1 Overview

Linear algebra is one of the most useful branches of applied mathematics for economists to invest in.

For example, many applied problems in economics and finance require the solution of a linear system of equations, such as

\[
\begin{align*}
y_1 &= ax_1 + bx_2 \\
y_2 &= cx_1 + dx_2
\end{align*}
\]

or, more generally,
The objective here is to solve for the unknowns \(x_1, \ldots, x_k\) \(\text{given} a_{11}, \ldots, a_{nk}\) and \(y_1, \ldots, y_n\). When considering such problems, it is essential that we first consider at least some of the following questions:

- Does a solution actually exist?
- Are there in fact many solutions, and if so how should we interpret them?
- If no solution exists, is there a best approximate solution?
- If a solution exists, how should we compute it?

These are the kinds of topics addressed by linear algebra.

In this lecture we will cover the basics of linear and matrix algebra, treating both theory and computation. We admit some overlap with this lecture, where operations on 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.

### 2.1.2 Vectors

A vector of length \(n\) is just a sequence (or array, or tuple) of \(n\) numbers, which we write as \(x = (x_1, \ldots, x_n)\) or \(x = [x_1, \ldots, x_n]\).

We will write these sequences either horizontally or vertically as we please.

(Later, when we wish to perform certain matrix operations, it will become necessary to distinguish between the two.)

The set of all \(n\)-vectors is denoted by \(\mathbb{R}^n\).

For example, \(\mathbb{R}^2\) is the plane, and a vector in \(\mathbb{R}^2\) is just a point in the plane.

Traditionally, vectors are represented visually as arrows from the origin to the point.

The following figure represents three vectors in this manner:

```latex
\begin{equation}
\begin{align*}
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{align*}
\end{equation}
```
vecs = ([2, 4], [-3, 3], [-4, -3.5])
x_vals = zeros(2, length(vecs))
y_vals = zeros(2, length(vecs))
labels = []

# Create matrices of x and y values, labels for plotting
for i = 1:length(vecs)
    v = vecs[i]
    x_vals[2, i] = v[1]
    y_vals[2, i] = v[2]
    labels = [labels; (1.1 * v[1], 1.1 * v[2], "$v")]
end
plot(x_vals, y_vals, arrow=true, color=:blue, legend=:none, xlims=(-5, 5), ylims=(-5, 5), annotations=labels, xticks=-5:1:5, yticks=-5:1:5, framestyle=:origin)
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

\[
x = \text{ones}(3)
\]

3-element Array\{Float64,1\}:

1.0
1.0
1.0

\[
y = [2, 4, 6]
\]

3-element Array\{Int64,1\}:

2
4
6

\[
x + y
\]
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
3-element Array{Float64,1}:
3.0
5.0
7.0

4x  # equivalent to 4 * x and 4 .* x

3-element Array{Float64,1}:
4.0
4.0
4.0

dot(x, y)  # Inner product of x and y
12.0

sum(x .* y)  # Gives the same result
12.0

norm(x)  # Norm of x
1.7320508075688772

sqrt(sum(x.^2))  # Gives the same result
1.7320508075688772
```
Span

Given a set of vectors \( A := \{a_1, \ldots, a_k\} \) in \( \mathbb{R}^n \), it's natural to think about the new vectors we can create by performing linear operations.

New vectors created in this manner are called \textit{linear combinations} of \( A \).

In particular, \( y \in \mathbb{R}^n \) is a linear combination of \( A := \{a_1, \ldots, a_k\} \) if

\[
y = \beta_1 a_1 + \cdots + \beta_k a_k \quad \text{for some scalars } \beta_1, \ldots, \beta_k
\]

In this context, the values \( \beta_1, \ldots, \beta_k \) are called the \textit{coefficients} of the linear combination.

The set of linear combinations of \( A \) is called the \textit{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.
plot!(x_vec, y_vec, z_vec, color=[:blue :red], linewidth=1.5, alpha=0.6, label=labels)

# Draw the plane
grid_size = 20
xr2 = linspace(x_min, x_max, grid_size)
yr2 = linspace(y_min, y_max, grid_size)
z2 = Array{Float64}(grid_size, grid_size)
for i in 1:grid_size
    for j in 1:grid_size
        z2[j, i] = f(xr2[i], yr2[j])
    end
end
surface!(xr2, yr2, z2, cbar=false, alpha=0.2, fill=:blues, xlims=(x_min, x_max), ylims=(x_min, x_max), zlims=(x_min, x_max), xticks=[0], yticks=[0], zticks=[0])
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 well see, it’s often desirable to find families of vectors with relatively large span, so that many vectors can be described by linear operators on a few vectors.

The condition we need for a set of vectors to have a large span is what’s called linear independence.

In particular, a collection of vectors $A := \{a_1, \ldots, a_k\}$ in $\mathbb{R}^n$ is said to be

- linearly dependent if some strict subset of $A$ has the same span as $A$
- linearly independent if it is not linearly dependent

Put differently, a set of vectors is linearly independent if no vector is redundant to the span, and linearly dependent otherwise.

To illustrate the idea, recall the figure that showed the span of vectors $\{a_1, a_2\}$ in $\mathbb{R}^3$ as a plane through the origin.

If we take a third vector $a_3$ and form the set $\{a_1, a_2, a_3\}$, this set will be

- linearly dependent if $a_3$ lies in the plane
- linearly independent otherwise

As another illustration of the concept, since $\mathbb{R}^n$ can be spanned by $n$ vectors (see the discussion of canonical basis vectors above), any collection of $m > n$ vectors in $\mathbb{R}^n$ must be linearly dependent.

The following statements are equivalent to linear independence of $A := \{a_1, \ldots, a_k\} \subset \mathbb{R}^n$

1. No vector in $A$ can be formed as a linear combination of the other elements.
2. If $\beta_1 a_1 + \cdots + \beta_k a_k = 0$ for scalars $\beta_1, \ldots, \beta_k$, then $\beta_1 = \cdots = \beta_k = 0$.

(The zero in the first expression is the origin of $\mathbb{R}^n$.)
Unique Representations

Another nice thing about sets of linearly independent vectors is that each element in the span has a unique representation as a linear combination of these vectors.

In other words, if \( A := \{a_1, \ldots, a_k\} \subset \mathbb{R}^n \) is linearly independent and

\[
y = \beta_1 a_1 + \cdots + \beta_k a_k
\]

then no other coefficient sequence \( \gamma_1, \ldots, \gamma_k \) will produce the same vector \( y \).

Indeed, if we also have \( y = \gamma_1 a_1 + \cdots + \gamma_k a_k \), then

\[
(\beta_1 - \gamma_1)a_1 + \cdots + (\beta_k - \gamma_k)a_k = 0
\]

Linear independence now implies \( \gamma_i = \beta_i \) for all \( i \)

2.1.3 Matrices

Matrices are a neat way of organizing data for use in linear operations.

An \( n \times k \) matrix is a rectangular array \( A \) of numbers with \( n \) rows and \( k \) columns:

\[
A = \begin{bmatrix}
a_{11} & a_{12} & \cdots & a_{1k} \\
a_{21} & a_{22} & \cdots & a_{2k} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nk}
\end{bmatrix}
\]

Often, the numbers in the matrix represent coefficients in a system of linear equations, as discussed at the start of this lecture.

For obvious reasons, the matrix \( A \) is also called a vector if either \( n = 1 \) or \( k = 1 \).

In the former case, \( A \) is called a row vector, while in the latter it is called a column vector.

If \( n = k \), then \( A \) is called square.

The matrix formed by replacing \( a_{ij} \) by \( a_{ji} \) for every \( i \) and \( j \) is called the transpose of \( A \), and denoted \( A' \) or \( A^\top \).

If \( A = A' \), then \( A \) is called symmetric.

For a square matrix \( A \), the \( i \) elements of the form \( a_{ii} \) for \( i = 1, \ldots, n \) are called the principal diagonal.

\( A \) is called diagonal if the only nonzero entries are on the principal diagonal.

If, in addition to being diagonal, each element along the principal diagonal is equal to 1, then \( A \) is called the identity matrix, and denoted by \( I \).
Matrix Operations

Just as was the case for vectors, a number of algebraic operations are defined for matrices.

Scalar multiplication and addition are immediate generalizations of the vector case:

\[ \gamma A = \gamma \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} = \begin{bmatrix} \gamma a_{11} & \cdots & \gamma a_{1k} \\ \vdots & \ddots & \vdots \\ \gamma a_{n1} & \cdots & \gamma a_{nk} \end{bmatrix} \]

and

\[ A + B = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} + \begin{bmatrix} b_{11} & \cdots & b_{1k} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nk} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1k} + b_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} + b_{n1} & \cdots & a_{nk} + b_{nk} \end{bmatrix} \]

In the latter case, the matrices must have the same shape in order for the definition to make sense.

We also have a convention for multiplying two matrices.

The rule for matrix multiplication generalizes the idea of inner products discussed above, and is designed to make multiplication play well with basic linear operations.

If \( A \) and \( B \) are two matrices, then their product \( AB \) is formed by taking as its \( i,j \)-th element the inner product of the \( i \)-th row of \( A \) and the \( j \)-th column of \( B \).

There are many tutorials to help you visualize this operation, such as this one, or the discussion on the Wikipedia page.

If \( A \) is \( n \times k \) and \( B \) is \( j \times m \), then to multiply \( A \) and \( B \) we require \( k = j \), and the resulting matrix \( AB \) is \( n \times m \).

As perhaps the most important special case, consider multiplying \( n \times k \) matrix \( A \) and \( k \times 1 \) column vector \( x \).

According to the preceding rule, this gives us an \( n \times 1 \) column vector

\[ A x = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + \cdots + a_{1k}x_k \\ \vdots \\ a_{n1}x_1 + \cdots + a_{nk}x_k \end{bmatrix} \quad (2.2) \]

Note: \( AB \) and \( BA \) are not generally the same thing.

Another important special case is the identity matrix.

You should check that if \( A \) is \( n \times k \) and \( I \) is the \( k \times k \) identity matrix, then \( AI = A \).

If \( I \) is the \( n \times n \) identity matrix, then \( IA = A \).
Matrices in 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

\[
A = \begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\]

2×2 Array{Int64,2}:
1 2
3 4

\[
\text{typeof}(A)
\]
Array{Int64,2}

\[
\text{size}(A)
\]
(2,2)

The \text{size} function returns a tuple giving the number of rows and columns

To get the transpose of \( A \), use \text{transpose}(A) or, more simply, \( A' \)

There are many convenient functions for creating common matrices (matrices of zeros, ones, etc.) see \textit{here}

Since operations are performed elementwise by default, scalar multiplication and addition have very natural syntax

\[
A = \text{eye}(3)
\]

3×3 Array{Float64,2}:
1.0 0.0 0.0
0.0 1.0 0.0
0.0 0.0 1.0

\[
B = \text{ones}(3, 3)
\]

3×3 Array{Float64,2}:
1.0 1.0 1.0
1.0 1.0 1.0
1.0 1.0 1.0

2\(A\)

3×3 Array{Float64,2}:
2.0 0.0 0.0
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, its known that $f$ is linear if and only if there exists a matrix $A$ such that $f(x) = Ax$ for all $x$

### 2.1.4 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

$$y = Ax \quad (2.3)$$

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
#=

@authors: Spencer Lyon <spencer.lyon@nyu.edu>
        Victoria Gregory <victoria.gregory@nyu.edu>
=#

f(x) = 0.6 * \cos(4.0 * x) + 1.3

xmin, xmax = -1.0, 1.0
Nx = 160
x = linspace(xmin, xmax, Nx)
y = f(x)
ya, yb = minimum(y), maximum(y)

pl = plot(x, y, color=:black, label=L"f(x)""
plot!(x, ya*ones(Nx, 1), fill_between=yb*ones(Nx, 1),
        fillalpha=0.1, color=:blue, label=""", lw=0)
plot!(zeros(2, 2), [ya ya; yb yb], lw=3, color=:blue, label=[L"range of f(x)"
        "\rightarrow"])
annotate!(0.04, -0.3, L"S05", ylins=(-0.6, 3.2))
vline!([0], color=:black, label="")
hline!([0], color=:black, label="")
plot!(foreground_color_axis=:white, foreground_color_text=:white,
        foreground_color_border=:white)

ybar = 1.5
plot!(x, x .* 0 .+ ybar, color=:black, linestyle=:dash, label="")
annotate!(0.05, 0.8 * ybar, L"y")

x_vals = Array{Float64}(2, 4)
y_vals = Array{Float64}(2, 4)
labels = []
for (i, z) in enumerate([-0.35, 0.35])
    x_vals[:, 2*i-1] = z*ones(2, 1)
y_vals[2, 2*i-1] = f(z)
    labels = [labels; (z, -0.2, LaTeXString("\$x_i\$"))]
end
plot!(x_vals, y_vals, color=:black, linestyle=:dash, label="",
        annotation=labels)

p2 = plot(x, y, color=:black, label=L"f(x)""
plot!(x, ya*ones(Nx, 1), fill_between=yb*ones(Nx, 1),
        fillalpha=0.1, color=:blue, label=""", lw=0)
plot!(zeros(2, 2), [ya ya; yb yb], lw=3, color=:blue, label=[L"range of f(x)"
        "\rightarrow"])
annotate!(0.04, -0.3, L"S05", ylins=(-0.6, 3.2))
vline!([0], color=:black, label="")
hline!([0], color=:black, label="")
plot!(foreground_color_axis=:white, foreground_color_text=:white,
        foreground_color_border=:white)
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 \( a_1, \ldots, a_k \) are the columns of \( A \), then
\[
Ax = x_1 a_1 + \cdots + x_k a_k
\]
Hence the range of \( f(x) = Ax \) is exactly the span of the columns of \( A \).
We want the range to be large, so that it contains arbitrary \( y \).
As you might recall, the condition that we want for the span to be large is **linear independence**.
A happy fact is that linear independence of the columns of \( A \) also gives us uniqueness.
Indeed, it follows from our *earlier discussion* that if \( \{a_1, \ldots, a_k\} \) are linearly independent and \( y = Ax = x_1 a_1 + \cdots + x_k a_k \), then no \( z \neq x \) satisfies \( y = Az \).

**The \( n \times n \) Case**

Let's discuss some more details, starting with the case where \( A \) is \( n \times n \).
This is the familiar case where the number of unknowns equals the number of equations.
For arbitrary \( y \in \mathbb{R}^n \), we hope to find a unique \( x \in \mathbb{R}^n \) such that \( y = Ax \).
In view of the observations immediately above, if the columns of \( A \) are linearly independent, then their span, and hence the range of \( f(x) = Ax \), is all of \( \mathbb{R}^n \).
Hence there always exists an \( x \) such that \( y = Ax \).
Moreover, the solution is unique.
In particular, the following are equivalent:
1. The columns of \( A \) are linearly independent.
2. For any \( y \in \mathbb{R}^n \), the equation \( y = Ax \) has a unique solution.
The property of having linearly independent columns is sometimes expressed as having **full column rank**.

**Inverse Matrices**

Can we give some sort of expression for the solution?
If \( y \) and \( A \) are scalar with \( A \neq 0 \), then the solution is \( x = A^{-1} y \).
A similar expression is available in the matrix case:
In particular, if square matrix \( A \) has full column rank, then it possesses a multiplicative **inverse matrix** \( A^{-1} \), with the property that \( AA^{-1} = A^{-1} A = I \).
As a consequence, if we pre-multiply both sides of \( y = Ax \) by \( A^{-1} \), we get \( x = A^{-1} y \).
This is the solution that we were looking for.
Determinants

Another quick comment about square matrices is that to every such matrix we assign a unique number called the determinant of the matrix. You can find the expression for it here. If the determinant of $A$ is not zero, then we say that $A$ is nonsingular.

Perhaps the most important fact about determinants is that $A$ is nonsingular if and only if $A$ is of full column rank.

This gives us a useful one-number summary of whether or not a square matrix can be inverted.

More Rows than Columns

This is the $n \times k$ case with $n > k$.

This case is very important in many settings, not least in the setting of linear regression (where $n$ is the number of observations, and $k$ is the number of explanatory variables).

Given arbitrary $y \in \mathbb{R}^n$, we seek an $x \in \mathbb{R}^k$ such that $y = Ax$.

In this setting, existence of a solution is highly unlikely.

Without much loss of generality, let’s go over the intuition focusing on the case where the columns of $A$ are linearly independent.

It follows that the span of the columns of $A$ is a $k$-dimensional subspace of $\mathbb{R}^n$.

This span is very unlikely to contain arbitrary $y \in \mathbb{R}^n$.

To see why, recall the figure above, where $k = 2$ and $n = 3$.

Imagine an arbitrarily chosen $y \in \mathbb{R}^3$, located somewhere in that three-dimensional space.

What’s the likelihood that $y$ lies in the span of $\{a_1, a_2\}$ (i.e., the two-dimensional plane through these points)?

In a sense it must be very small, since this plane has zero thickness.

As a result, in the $n > k$ case we usually give up on existence.

However, we can still seek a best approximation, for example an $x$ that makes the distance $\|y - Ax\|$ as small as possible.

To solve this problem, one can use either calculus or the theory of orthogonal projections.

The solution is known to be $\hat{x} = (A' A)^{-1} A' y$. See for example chapter 3 of these notes.

More Columns than Rows

This is the $n \times k$ case with $n < k$, so there are fewer equations than unknowns.

In 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 $A$ consist of 3 vectors in $\mathbb{R}^2$. 

2.1. Linear Algebra
This set can never be linearly independent, since it is possible to find two vectors that span $\mathbb{R}^2$ (For example, use the canonical basis vectors)

It follows that one column is a linear combination of the other two

For example, lets 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];
y = ones(2, 1);  # A column vector

2x1 Array{Float64,2}:
1.0
1.0

det(A)

-2.0

A_inv = inv(A)

2x2 Array{Float64,2}:
-2.0 1.0
 1.5 -0.5

x = A_inv * y  # solution

2x1 Array{Float64,2}:
-1.0
1.0

A * x  # should equal y (a vector of ones)

2x1 Array{Float64,2}:
1.0
1.0
```
# produces the same solution

2×1 Array{Float64,2}:
-1.0
  1.0

Observe how we can solve for \( x = A^{-1}y \) by either via \( \text{inv}(A) \times y \), or using \( A \ \backslash \ y \)

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 \( A \ \backslash \ y \) returns the least squares solution \( \hat{x} = (A'A)^{-1}A'y \).

## 2.1.5 Eigenvalues and Eigenvectors

Let \( A \) be an \( n \times n \) square matrix.

If \( \lambda \) is scalar and \( v \) is a non-zero vector in \( \mathbb{R}^n \) such that

\[
Av = \lambda v
\]

then we say that \( \lambda \) is an eigenvalue of \( A \), and \( v \) is an eigenvector.

Thus, an eigenvector of \( A \) is a vector such that when the map \( f(x) = Ax \) is applied, \( v \) is merely scaled.

The next figure shows two eigenvectors (blue arrows) and their images under \( A \) (red arrows).

As expected, the image \( Av \) of each \( v \) is just a scaled version of the original.

```julia
A = [1 2
     2 1]
evals, evecs = eig(A)
a1, a2 = evals[1], evals[2]
evecs = evecs[:, 1], evecs[:, 2]
eig_1 = zeros(2, length(evecs))
eig_2 = zeros(2, length(evecs))
labels = []

for i = 1:length(evecs)
    v = evecs[i]
    eig_1[2, i] = v[1]
    eig_2[2, i] = v[2]
end

x = linspace(-5, 5, 10)
y = -linspace(-5, 5, 10)

plot(eig_1[:, 2], a1 * eig_2[:, 2], arrow=true, color=:red,
     legend=:none, xlims=(-3, 3), ylims=(-3, 3),
     annotations=labels, xticks=-5:1:5, yticks=-5:1:5,
     framestyle=:origin)
plot!(a2 * eig_1[:, 2], a2 * eig_2, arrow=true, color=:red)
plot!(eig_1, eig_2, arrow=true, color=:blue)
```
The eigenvalue equation is equivalent to \((A - \lambda I)v = 0\), and this has a nonzero solution \(v\) only when the columns of \(A - \lambda I\) are linearly dependent.

This in turn is equivalent to stating that the determinant is zero.

Hence to find all eigenvalues, we can look for \(\lambda\) such that the determinant of \(A - \lambda I\) is zero.

This problem can be expressed as one of solving for the roots of a polynomial in \(\lambda\) of degree \(n\).

This in turn implies the existence of \(n\) solutions in the complex plane, although some might be repeated.

Some nice facts about the eigenvalues of a square matrix \(A\) are as follows:

1. The determinant of \(A\) equals the product of the eigenvalues.
2. The trace of \(A\) (the sum of the elements on the principal diagonal) equals the sum of the eigenvalues.
3. If \(A\) is symmetric, then all of its eigenvalues are real.
4. If \(A\) is invertible and \(\lambda_1, \ldots, \lambda_n\) are its eigenvalues, then the eigenvalues of \(A^{-1}\) are \(1/\lambda_1, \ldots, 1/\lambda_n\).
A corollary of the first statement is that a matrix is invertible if and only if all its eigenvalues are nonzero.

Using Julia, we can solve for the eigenvalues and eigenvectors of a matrix as follows:

```julia
A = [1.0 2.0; 2.0 1.0];
evals, evecs = eig(A);
```

```plaintext
evals
2-element Array{Float64,1}:
  -1.0
  3.0

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.

### 2.1.6 Further Topics

We round out our discussion by briefly mentioning several other important topics.

### Series Expansions

Recall the usual summation formula for a geometric progression, which states that if $|a| < 1$, then

$$\sum_{k=0}^{\infty} a^k = (1 - a)^{-1}$$

A generalization of this idea exists in the matrix setting.
Matrix Norms

Let $A$ be a square matrix, and let

$$\|A\| := \max_{\|x\|=1} \|Ax\|$$

The norms on the right-hand side are ordinary vector norms, while the norm on the left-hand side is a matrix norm in this case, the so-called spectral norm.

For example, for a square matrix $S$, the condition $\|S\| < 1$ means that $S$ is contractive, in the sense that it pulls all vectors towards the origin.

**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

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k$$  \hspace{1cm} (2.4)

**Spectral Radius**

A result known as Gelfand's formula tells us that, for any square matrix $A$,

$$\rho(A) = \lim_{k \to \infty} \|A^k\|^{1/k}$$

Here $\rho(A)$ is the spectral radius, defined as $\max_i |\lambda_i|$, where $\{\lambda_i\}$ 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 $x \in \mathbb{R}^n$

---

1 Suppose that $\|S\| < 1$. Take any nonzero vector $x$, and let $r := \|x\|$. We have $\|Sx\| = r\|S(x/r)\| \leq r\|S\| < r = \|x\|$. Hence every point is pulled towards the origin.
Analogous definitions exist for negative definite and negative semi-definite matrices.

It is notable that if $A$ is positive definite, then all of its eigenvalues are strictly positive, and hence $A$ is invertible (with positive definite inverse).

### Differentiating Linear and Quadratic forms

The following formulas are useful in many economic contexts. Let

- $z, x$ and $a$ all be $n \times 1$ 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. $\frac{\partial y'Bz}{\partial B} = y'z'$

Exercise 1 below asks you to apply these formulas.

### Further Reading

The documentation of the linear algebra features built into Julia can be found [here](#).

Chapters 2 and 3 of the *Econometric Theory* contains a discussion of linear algebra along the same lines as above, with solved exercises.

If you don’t mind a slightly abstract approach, a nice intermediate-level text on linear algebra is [Janich94](#).

### 2.1.7 Exercises

**Exercise 1**

Let $x$ be a given $n \times 1$ vector and consider the problem

$$v(x) = \max_{y,u} \left\{ -y'Py - u'Qu \right\}$$

subject to the linear constraint

$$y = Ax + Bu$$

Here

- $P$ is an $n \times n$ matrix and $Q$ is an $m \times m$ matrix.
• $A$ is an $n \times n$ matrix and $B$ is an $n \times m$ matrix

• both $P$ and $Q$ are symmetric and positive semidefinite

(What must the dimensions of $y$ and $u$ be to make this a well-posed problem?)

One way to solve the problem is to form the Lagrangian

$$\mathcal{L} = -y'Py - u'Qu + \lambda' [Ax + Bu - y]$$

where $\lambda$ is an $n \times 1$ vector of Lagrange multipliers

Try applying the formulas given above for differentiating quadratic and linear forms to obtain the first-order conditions for maximizing $\mathcal{L}$ 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 dont care about the Lagrange multipliers, we can substitute the constraint into the objective function, and then just maximize $-(Ax + Bu)'P(Ax + Bu) - u'Qu$ with respect to $u$. You can verify that this leads to the same maximizer.

### 2.1.8 Solutions

Thanks to Willem Hekman and Guanlong Ren for providing this solution.

**Exercise 1**

We have an optimization problem:

$$v(x) = \max_{y,u} \{-y'Py - u'Qu\}$$

s.t.

$$y = Ax + Bu$$

with primitives

• $P$ be a symmetric and positive semidefinite $n \times n$ matrix.

• $Q$ be a symmetric and positive semidefinite $m \times m$ matrix.

• $A$ an $n \times n$ matrix.

• $B$ an $n \times m$ matrix.
The associated Lagrangian is:

\[ L = -y'Py - u'Qu + \lambda'[Ax + Bu - y] \]

1.

Differentiating Lagrangian equation w.r.t. \( y \) and setting its derivative equal to zero yields

\[
\frac{\partial L}{\partial y} = -(P + P')y - \lambda = -2Py - \lambda = 0,
\]

since \( P \) is symmetric.

Accordingly, the first-order condition for maximizing \( L \) w.r.t. \( y \) implies

\[ \lambda = -2Py. \]

2.

Differentiating Lagrangian equation w.r.t. \( u \) and setting its derivative equal to zero yields

\[
\frac{\partial L}{\partial u} = -(Q + Q')u - B'\lambda = -2Qu + B'\lambda = 0.
\]

Substituting \( \lambda = -2Py \) gives

\[ Qu + B'Py = 0. \]

Substituting the linear constraint \( y = Ax + Bu \) into above equation gives

\[ Qu + B'P(Ax + Bu) = 0 \]

\[ (Q + B'PB)u + B'PAx = 0 \]

which is the first-order condition for maximizing \( L \) w.r.t. \( u \).

Thus, the optimal choice of \( u \) must satisfy

\[ u = -(Q + B'PB)^{-1}B'PAx, \]

which follows from the definition of the first-order conditions for Lagrangian equation.

3.

Rewriting our problem by substituting the constraint into the objective function, we get

\[ v(x) = \max_u \{ -(Ax + Bu)'P(Ax + Bu) - u'Qu \}. \]
Since we know the optimal choice of $u$ satisfies $u = -(Q + BPB)^{-1}BPAx$, then

$$v(x) = -(Ax + Bu)'P(Ax + Bu) - u'Qu \quad \text{with} \quad u = -(Q + B'PB)^{-1}B'PAx$$

To evaluate the function

$$v(x) = -(Ax + Bu)'P(Ax + Bu)$$

$$\quad = -(x'A' + u'B')P(Ax + Bu) - u'Qu$$

$$\quad = -x'A'PAx - u'B'PAx - x'A'PBu - u'B'PBu - u'Qu$$

$$\quad = -x'A'PAx - 2u'B'PAx - u'(Q + B'PB)u$$

For simplicity, denote by $S := (Q + B'PB)^{-1}B'PA$, then $u = -Sx$.

Regarding the second term $-2u'B'PAx$,

$$-2u'B'PAx = -2x'S'B'PAx$$

$$\quad = 2x'A'PB(Q + B'PB)^{-1}B'PAx$$

Notice that the term $(Q + B'PB)^{-1}$ is symmetric as both $P$ and $Q$ are symmetric.

Regarding the third term $-u'(Q + B'PB)u$,

$$-u'(Q + B'PB)u = -x'S'(Q + B'PB)Sx$$

$$\quad = -x'A'PB(Q + B'PB)^{-1}B'PAx$$

Hence, the summation of second and third terms is $x'A'PB(Q + B'PB)^{-1}B'PAx$.

This implies that

$$v(x) = -x'A'PAx - 2u'B'PAx - u'(Q + B'PB)u$$

$$\quad = -x'A'PAx + x'A'PB(Q + B'PB)^{-1}B'PAx$$

$$\quad = -x'[A'PA - A'PB(Q + B'PB)^{-1}B'PA]x$$

Therefore, the solution to the optimization problem $v(x) = -x'\hat{P}x$ follows the above result by denoting $\hat{P} := A'PA - A'PB(Q + B'PB)^{-1}B'PA$.

### 2.2 Orthogonal Projections and Their Applications

#### Contents

- **Orthogonal Projections and Their Applications**
  - Overview
  - Key Definitions
  - The Orthogonal Projection Theorem
  - Orthonormal Basis
2.2.1 Overview

Orthogonal projection is a cornerstone of vector space methods, with many diverse applications. These include, but are not limited to,

- Least squares projection, also known as linear regression
- Conditional expectations for multivariate normal (Gaussian) distributions
- Gram–Schmidt orthogonalization
- QR decomposition
- Orthogonal polynomials
  - etc

In this lecture we focus on

- key ideas
- least squares regression

Further Reading

For background and foundational concepts, see our lecture on linear algebra.

For more proofs and greater theoretical detail, see A Primer in Econometric Theory.

For a complete set of proofs in a general setting, see, for example, [Rom05].

For an advanced treatment of projection in the context of least squares prediction, see this book chapter.

2.2.2 Key Definitions

Assume $x, z \in \mathbb{R}^n$

Define $\langle x, z \rangle = \sum_i x_i z_i$

Recall $\|x\|^2 = \langle x, x \rangle$

The law of cosines states that $\langle x, z \rangle = \|x\| \|z\| \cos(\theta)$ where $\theta$ is the angle between the vectors $x$ and $z$.

When $\langle x, z \rangle = 0$, then $\cos(\theta) = 0$ and $x$ and $z$ are said to be orthogonal and we write $x \perp z$. 

2.2. Orthogonal Projections and Their Applications
For a linear subspace $S \subseteq \mathbb{R}^n$, we call $x \in \mathbb{R}^n$ orthogonal to $S$ if $x \perp z$ for all $z \in S$, and write $x \perp S$.
The orthogonal complement of linear subspace $S \subset \mathbb{R}^n$ is the set $S^\perp := \{x \in \mathbb{R}^n : x \perp S\}$.
$S^\perp$ is a linear subspace of $\mathbb{R}^n$

- To see this, fix $x, y \in S^\perp$ and $\alpha, \beta \in \mathbb{R}$
- Observe that if $z \in S$, then
  \[
  \langle \alpha x + \beta y, z \rangle = \alpha \langle x, z \rangle + \beta \langle y, z \rangle = \alpha \times 0 + \beta \times 0 = 0
  \]
- Hence $\alpha x + \beta y \in S^\perp$, as was to be shown

A set of vectors $\{x_1, \ldots, x_k\} \subset \mathbb{R}^n$ is called an orthogonal set if $x_i \perp x_j$ whenever $i \neq j$

If $\{x_1, \ldots, x_k\}$ is an orthogonal set, then the Pythagorean Law states that

\[
\|x_1 + \cdots + x_k\|^2 = \|x_1\|^2 + \cdots + \|x_k\|^2
\]

For example, when $k = 2$, $x_1 \perp x_2$ implies

\[
\|x_1 + x_2\|^2 = \langle x_1 + x_2, x_1 + x_2 \rangle = \langle x_1, x_1 \rangle + 2\langle x_2, x_1 \rangle + \langle x_2, x_2 \rangle = \|x_1\|^2 + \|x_2\|^2
\]

**Linear Independence vs Orthogonality**

If $X \subset \mathbb{R}^n$ is an orthogonal set and $0 \notin X$, then $X$ is linearly independent

Proving this is a nice exercise

While the converse is not true, a kind of partial converse holds, as well *see below*
2.2.3 The Orthogonal Projection Theorem

What vector within a linear subspace of $\mathbb{R}^n$ best approximates a given vector in $\mathbb{R}^n$?

The next theorem provides answers to this question.

**Theorem (OPT)** Given $y \in \mathbb{R}^n$ and linear subspace $S \subset \mathbb{R}^n$, there exists a unique solution to the minimization problem

$$\hat{y} := \arg \min_{z \in S} \|y - z\|$$

The minimizer $\hat{y}$ is the unique vector in $\mathbb{R}^n$ that satisfies

- $\hat{y} \in S$
- $y - \hat{y} \perp S$

The vector $\hat{y}$ is called the **orthogonal projection** of $y$ onto $S$.

The next figure provides some intuition.

**Proof of sufficiency**

Well omit the full proof.

But we will prove sufficiency of the asserted conditions.
To this end, let \( y \in \mathbb{R}^n \) and let \( S \) be a linear subspace of \( \mathbb{R}^n \).

Let \( \hat{y} \) be a vector in \( \mathbb{R}^n \) such that \( \hat{y} \in S \) and \( y - \hat{y} \perp S \).

Let \( z \) be any other point in \( S \) and use the fact that \( S \) is a linear subspace to deduce

\[
\|y - z\|^2 = \|(y - \hat{y}) + (\hat{y} - z)\|^2 = \|y - \hat{y}\|^2 + \|\hat{y} - z\|^2
\]

Hence \( \|y - z\| \geq \|y - \hat{y}\| \), which completes the proof.

**Orthogonal Projection as a Mapping**

For a linear space \( Y \) and a fixed linear subspace \( S \), we have a functional relationship

\[
y \in Y \mapsto \text{its orthogonal projection } \hat{y} \in S
\]

By the OPT, this is a well-defined mapping or operator from \( \mathbb{R}^n \) to \( \mathbb{R}^n \).

In what follows we denote this operator by a matrix \( P \).

- \( Py \) represents the projection \( \hat{y} \).
- This is sometimes expressed as \( \hat{E}_S y = Py \), where \( \hat{E} \) denotes a wide-sense expectations operator and the subscript \( S \) indicates that we are projecting \( y \) onto the linear subspace \( S \).

The operator \( P \) is called the **orthogonal projection mapping onto** \( S \).

It is immediate from the OPT that for any \( y \in \mathbb{R}^n \)
1. \(Py \in S\) and 
2. \(y - Py \perp S\)

From this we can deduce additional useful properties, such as

1. \(\|y\|^2 = \|Py\|^2 + \|y - Py\|^2\) and 
2. \(\|Py\| \leq \|y\|\)

For example, to prove 1, observe that \(y = Py + y - Py\) and apply the Pythagorean law

**Orthogonal Complement**

Let \(S \subset \mathbb{R}^n\).

The **orthogonal complement** of \(S\) is the linear subspace \(S^\perp\) that satisfies \(x_1 \perp x_2\) for every \(x_1 \in S\) and \(x_2 \in S^\perp\)

Let \(Y\) be a linear space with linear subspace \(S\) and its orthogonal complement \(S^\perp\).

We write

\[
Y = S \oplus S^\perp
\]

to indicate that for every \(y \in Y\) there is unique \(x_1 \in S\) and a unique \(x_2 \in S^\perp\) such that \(y = x_1 + x_2\).

Moreover, \(x_1 = \hat{E}_Sy\) and \(x_2 = y - \hat{E}_Sy\)

This amounts to another version of the OPT:

**Theorem.** If \(S\) is a linear subspace of \(\mathbb{R}^n\), \(\hat{E}_Sy = Py\) and \(\hat{E}_{S^\perp}y = My\), then

\[
Py \perp My \quad \text{and} \quad y = Py + My \quad \text{for all} \; y \in \mathbb{R}^n
\]

The next figure illustrates
2.2.4 Orthonormal Basis

An orthogonal set of vectors $O \subset \mathbb{R}^n$ is called an **orthonormal set** if $\|u\| = 1$ for all $u \in O$.

Let $S$ be a linear subspace of $\mathbb{R}^n$ and let $O \subset S$.

If $O$ is orthonormal and $\text{span} \ O = S$, then $O$ is called an **orthonormal basis** of $S$.

$O$ is necessarily a basis of $S$ (being independent by orthogonality and the fact that no element is the zero vector).

One example of an orthonormal set is the canonical basis $\{e_1, \ldots, e_n\}$ that forms an orthonormal basis of $\mathbb{R}^n$, where $e_i$ is the $i$ th unit vector.

If $\{u_1, \ldots, u_k\}$ is an orthonormal basis of linear subspace $S$, then

$$x = \sum_{i=1}^{k} \langle x, u_i \rangle u_i \quad \text{for all} \quad x \in S$$

To see this, observe that since $x \in \text{span}\{u_1, \ldots, u_k\}$, we can find scalars $\alpha_1, \ldots, \alpha_k$ that verify

$$x = \sum_{j=1}^{k} \alpha_j u_j$$

(2.16)
Taking the inner product with respect to \( u_i \) gives

\[
\langle x, u_i \rangle = \sum_{j=1}^{k} \alpha_j \langle u_j, u_i \rangle = \alpha_i
\]

Combining this result with (2.16) verifies the claim

**Projection onto an Orthonormal Basis**

When the subspace onto which are projecting is orthonormal, computing the projection simplifies:

**Theorem** If \( \{u_1, \ldots, u_k\} \) is an orthonormal basis for \( S \), then

\[
P y = \sum_{i=1}^{k} \langle y, u_i \rangle u_i, \quad \forall y \in \mathbb{R}^n
\]  

(2.17)

Proof: Fix \( y \in \mathbb{R}^n \) and let \( P y \) be defined as in (2.17)

Clearly, \( P y \in S \)

We claim that \( y - P y \perp S \) also holds

It suffices to show that \( y - P y \perp \) any basis vector \( u_i \) (why?)

This is true because

\[
\left\langle y - \sum_{i=1}^{k} \langle y, u_i \rangle u_i, u_j \right\rangle = \langle y, u_j \rangle - \sum_{i=1}^{k} \langle y, u_i \rangle \langle u_i, u_j \rangle = 0
\]

2.2.5 Projection Using Matrix Algebra

Let \( S \) be a linear subspace of \( \mathbb{R}^n \) and let \( y \in \mathbb{R}^n \).

We want to compute the matrix \( P \) that verifies

\[
\hat{E}_S y = P y
\]

Evidently \( P y \) is a linear function from \( y \in \mathbb{R}^n \) to \( P y \in \mathbb{R}^n \)

This reference is useful [https://en.wikipedia.org/wiki/Linear_map#Matrices](https://en.wikipedia.org/wiki/Linear_map#Matrices)

**Theorem.** Let the columns of \( n \times k \) matrix \( X \) form a basis of \( S \). Then

\[
P = X (X'X)^{-1} X'
\]

Proof: Given arbitrary \( y \in \mathbb{R}^n \) and \( P = X (X'X)^{-1} X' \), our claim is that

1. \( P y \in S \), and
2. \( y - P y \perp S \)
Claim 1 is true because

\[ Py = X(X'X)^{-1}X'y = Xa \quad \text{when} \quad a := (X'X)^{-1}X'y \]

An expression of the form \( Xa \) is precisely a linear combination of the columns of \( X \), and hence an element of \( S \).

Claim 2 is equivalent to the statement

\[ y - X(X'X)^{-1}X'y \perp Xb \quad \text{for all} \quad b \in \mathbb{R}^K \]

This is true: If \( b \in \mathbb{R}^K \), then

\[ (Xb)'[y - X(X'X)^{-1}X'y] = b'[X'y - X'y] = 0 \]

The proof is now complete.

**Starting with \( X \)**

It is common in applications to start with \( n \times k \) matrix \( X \) with linearly independent columns and let

\[ S := \text{span} \{ \text{col}_1 X, \ldots, \text{col}_k X \} \]

Then the columns of \( X \) form a basis of \( S \).

From the preceding theorem, \( P = X(X'X)^{-1}X'y \) projects \( y \) onto \( S \).

In this context, \( P \) is often called the **projection matrix**

- The matrix \( M = I - P \) satisfies \( My = \hat{E}_{S^\perp}y \) and is sometimes called the **annihilator matrix**

**The Orthonormal Case**

Suppose that \( U \) is \( n \times k \) with orthonormal columns

Let \( u_i := \text{col}_i U \) for each \( i \), let \( S := \text{span} U \) and let \( y \in \mathbb{R}^n \)

We know that the projection of \( y \) onto \( S \) is

\[ Py = U(U'U)^{-1}U'y \]

Since \( U \) has orthonormal columns, we have \( U'U = I \)

Hence

\[ Py = UU'y = \sum_{i=1}^{k} \langle u_i, y \rangle u_i \]

We have recovered our earlier result about projecting onto the span of an orthonormal basis.
Application: Overdetermined Systems of Equations

Let $y \in \mathbb{R}^n$ and let $X$ is $n \times k$ with linearly independent columns.

Given $X$ and $y$, we seek $b \in \mathbb{R}^k$ satisfying the system of linear equations $Xb = y$.

If $n > k$ (more equations than unknowns), then $b$ is said to be **overdetermined**.

Intuitively, we may not be able find a $b$ that satisfies all $n$ equations.

The best approach here is to:

- Accept that an exact solution may not exist.
- Look instead for an approximate solution.

By approximate solution, we mean a $b \in \mathbb{R}^k$ such that $Xb$ is as close to $y$ as possible.

The next theorem shows that the solution is well defined and unique.

The proof uses the OPT.

**Theorem** The unique minimizer of $\|y - Xb\|$ over $b \in \mathbb{R}^K$ is

$$\hat{\beta} := (X'X)^{-1}X'y$$

Proof: Note that

$$X\hat{\beta} = X(X'X)^{-1}X'y = Py$$

Since $Py$ is the orthogonal projection onto $\text{span}(X)$ we have

$$\|y - Py\| \leq \|y - z\| \text{ for any } z \in \text{span}(X)$$

Because $Xb \in \text{span}(X)$

$$\|y - X\hat{\beta}\| \leq \|y - Xb\| \text{ for any } b \in \mathbb{R}^K$$

This is what we aimed to show.

---

### 2.2.6 Least Squares Regression

Let’s apply the theory of orthogonal projection to least squares regression.

This approach provides insights about many geometric properties of linear regression.

We treat only some examples.

**Squared risk measures**

Given pairs $(x, y) \in \mathbb{R}^K \times \mathbb{R}$, consider choosing $f : \mathbb{R}^K \rightarrow \mathbb{R}$ to minimize the **risk**

$$R(f) := \mathbb{E} [(y - f(x))^2]$$
If probabilities and hence $\mathbb{E}$ are unknown, we cannot solve this problem directly. However, if a sample is available, we can estimate the risk with the empirical risk:

$$\min_{f \in \mathcal{F}} \frac{1}{N} \sum_{n=1}^{N} (y_n - f(x_n))^2$$

Minimizing this expression is called empirical risk minimization. The set $\mathcal{F}$ is sometimes called the hypothesis space. The theory of statistical learning tells us that to prevent overfitting we should take the set $\mathcal{F}$ to be relatively simple. If we let $\mathcal{F}$ be the class of linear functions $1/N$, the problem is

$$\min_{b \in \mathbb{R}^K} \sum_{n=1}^{N} (y_n - b' x_n)^2$$

This is the sample linear least squares problem.

**Solution**

Define the matrices

$$y := \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}, \quad x_n := \begin{pmatrix} x_{n1} \\ x_{n2} \\ \vdots \\ x_{nK} \end{pmatrix} = n\text{-th obs on all regressors}$$

and

$$X := \begin{pmatrix} x_1' \\ x_2' \\ \vdots \\ x_N' \end{pmatrix} := \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1K} \\ x_{21} & x_{22} & \cdots & x_{2K} \\ \vdots & \vdots & \vdots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{NK} \end{pmatrix}$$

We assume throughout that $N > K$ and $X$ is full column rank. If you work through the algebra, you will be able to verify that $\|y - Xb\|^2 = \sum_{n=1}^{N} (y_n - b' x_n)^2$.

Since monotone transforms don’t affect minimizers, we have

$$\arg \min_{b \in \mathbb{R}^K} \sum_{n=1}^{N} (y_n - b' x_n)^2 = \arg \min_{b \in \mathbb{R}^K} \|y - Xb\|$$

By our results about overdetermined linear systems of equations, the solution is

$$\hat{\beta} := (X'X)^{-1} X'y$$

Let $P$ and $M$ be the projection and annihilator associated with $X$:

$$P := X(X'X)^{-1} X' \quad \text{and} \quad M := I - P$$
The vector of fitted values is
\[ \hat{y} := X\hat{\beta} = Py \]

The vector of residuals is
\[ \hat{u} := y - \hat{y} = y - Py = My \]

Here are some more standard definitions:

- The total sum of squares is \( \|y\|^2 \)
- The sum of squared residuals is \( \|\hat{u}\|^2 \)
- The explained sum of squares is \( \|\hat{y}\|^2 \)

\[ \text{TSS} = \text{ESS} + \text{SSR} \]

We can prove this easily using the OPT

From the OPT we have \( y = \hat{y} + \hat{u} \) and \( \hat{u} \perp \hat{y} \)

Applying the Pythagorean law completes the proof

### 2.2.7 Orthogonalization and Decomposition

Let's return to the connection between linear independence and orthogonality touched on above

A result of much interest is a famous algorithm for constructing orthonormal sets from linearly independent sets

The next section gives details

**Gram-Schmidt Orthogonalization**

**Theorem** For each linearly independent set \( \{x_1, \ldots, x_k\} \subset \mathbb{R}^n \), there exists an orthonormal set \( \{u_1, \ldots, u_k\} \) with

\[ \text{span}\{x_1, \ldots, x_i\} = \text{span}\{u_1, \ldots, u_i\} \text{ for } i = 1, \ldots, k \]

The **Gram-Schmidt orthogonalization** procedure constructs an orthogonal set \( \{u_1, u_2, \ldots, u_n\} \)

One description of this procedure is as follows:

- For \( i = 1, \ldots, k \), form \( S_i := \text{span}\{x_1, \ldots, x_i\} \) and \( S_i^\perp \)
- Set \( v_1 = x_1 \)
- For \( i \geq 2 \) set \( v_i := E_{S_{i-1}^\perp} x_i \) and \( u_i := v_i/\|v_i\| \)

The sequence \( u_1, \ldots, u_k \) has the stated properties

A Gram-Schmidt orthogonalization construction is a key idea behind the Kalman filter described in *A First Look at the Kalman filter*

In some exercises below you are asked to implement this algorithm and test it using projection
QR Decomposition

The following result uses the preceding algorithm to produce a useful decomposition.

**Theorem** If $X$ is $n \times k$ with linearly independent columns, then there exists a factorization $X = QR$ where

- $R$ is $k \times k$, upper triangular, and nonsingular
- $Q$ is $n \times k$ with orthonormal columns

Proof sketch: Let

- $x_j := \text{col}_j(X)$
- $\{u_1, \ldots, u_k\}$ be orthonormal with same span as $\{x_1, \ldots, x_k\}$ (to be constructed using Gram–Schmidt)
- $Q$ be formed from cols $u_i$

Since $x_j \in \text{span}\{u_1, \ldots, u_j\}$, we have

$$ x_j = \sum_{i=1}^{j} \langle u_i, x_j \rangle u_i \quad \text{for } j = 1, \ldots, k $$

Some rearranging gives $X = QR$

**Linear Regression via QR Decomposition**

For matrices $X$ and $y$ that overdetermine $\beta$ in the linear equation system $y = X\beta$, we found the least squares approximator $\hat{\beta} = (X'X)^{-1}X'y$

Using the QR decomposition $X = QR$ gives

$$ \hat{\beta} = (R'Q'R)^{-1}R'Q'y $$

$$ = (R'R)^{-1}R'Q'y $$

$$ = R^{-1}(R')^{-1}R'Q'y = R^{-1}Q'y $$

Numerical routines would in this case use the alternative form $R\hat{\beta} = Q'y$ and back substitution

**2.2.8 Exercises**

**Exercise 1**

Show that, for any linear subspace $S \subset \mathbb{R}^n$, $S \cap S^\perp = \{0\}$

**Exercise 2**

Let $P = X(X'X)^{-1}X'$ and let $M = I - P$. Show that $P$ and $M$ are both idempotent and symmetric. Can you give any intuition as to why they should be idempotent?
Exercise 3

Using Gram-Schmidt orthogonalization, produce a linear projection of $y$ onto the column space of $X$ and verify this using the projection matrix $P := X(X'X)^{-1}X'$ and also using QR decomposition, where:

$$y := \begin{pmatrix} 1 \\ 3 \\ -3 \end{pmatrix},$$

and

$$X := \begin{pmatrix} 1 & 0 \\ 0 & -6 \\ 2 & 2 \end{pmatrix}$$

2.2.9 Solutions

Exercise 1

If $x \in S$ and $x \in S^\perp$, then we have in particular that $\langle x, x \rangle = 0$. But then $x = 0$.

Exercise 2

Symmetry and idempotence of $M$ and $P$ can be established using standard rules for matrix algebra. The intuition behind idempotence of $M$ and $P$ is that both are orthogonal projections. After a point is projected into a given subspace, applying the projection again makes no difference. (A point inside the subspace is not shifted by orthogonal projection onto that space because it is already the closest point in the subspace to itself.)

Exercise 3

Here is a function that computes the orthonormal vectors using the GS algorithm given in the lecture.

```julia
""" Implements Gram-Schmidt orthogonalization.

Parameters
-------
X : an n x k array with linearly independent columns

Returns
-------
U : an n x k array with orthonormal columns
"""

function gram_schmidt(X)
    n, k = size(X)
```

2.2. Orthogonal Projections and Their Applications
U = \textbf{Array}(\texttt{Float64})(n, k)
I = \text{eye}(n)

# The first col of U is just the normalized first col of X
v1 = X[:,1]
U[:,1] = v1 / \text{norm}(v1)

for i in 2:k
    # Set up
    b = X[:,i]  # The vector we're going to project
    Z = X[:, 1:i-1]  # first i-1 columns of X

    # Project onto the orthogonal complement of the col span of Z
    M = I - Z * inv(Z' * Z) * Z'
    u = M * b

    # Normalize
    U[:,i] = u / \text{norm}(u)
end

\textbf{return} U
end

\textbf{gram_schmidt}

Here are the arrays well work with

\[
y = [1 \ 3 \ -3]' \\
X = [1 \ 0; \ 0 \ -6; \ 2 \ 2];
\]

First lets do ordinary projection of \(y\) onto the basis spanned by the columns of \(X\).

\[
Py1 = X * \text{inv}(X' * X) * X' * y
\]

\[
\begin{array}{c}
3\times 1 \ \text{Array}(\texttt{Float64},2):\\
-0.565217 \\
3.26087 \\
-2.21739
\end{array}
\]

Now lets orthogonalize first, using Gram–Schmidt:

\[
U = \text{gram_schmidt}(X)
\]

\[
\begin{array}{cc}
3\times 2 \ \text{Array}(\texttt{Float64},2):\\
0.447214 & -0.131876 \\
0.0 & -0.989071 \\
0.894427 & 0.065938
\end{array}
\]

Now we can project using the orthonormal basis and see if we get the same thing:
The result is the same. To complete the exercise, we get an orthonormal basis by QR decomposition and project once more.

\[
Q, R = \text{qr}(X)
\]

\[
\begin{pmatrix}
-0.447214 & -0.131876; 0.0 & -0.989071; -0.894427 & 0.065938 \\
-2.23607 & -1.78885; 0.0 & 6.0663
\end{pmatrix}
\]

Again, the result is the same

### 2.3 LLN and CLT

**Contents**

- **LLN and CLT**
  - Overview
  - Relationships
  - LLN
  - CLT
  - Exercises
  - Solutions

#### 2.3.1 Overview

This lecture illustrates two of the most important theorems of probability and statistics: The law of large numbers (LLN) and the central limit theorem (CLT)
These beautiful theorems lie behind many of the most fundamental results in econometrics and quantitative economic modeling.

The lecture is based around simulations that show the LLN and CLT in action.

We also demonstrate how the LLN and CLT break down when the assumptions they are based on do not hold.

In addition, we examine several useful extensions of the classical theorems, such as

- The delta method, for smooth functions of random variables
- The multivariate case

Some of these extensions are presented as exercises.

### 2.3.2 Relationships

The CLT refines the LLN.

The LLN gives conditions under which sample moments converge to population moments as sample size increases.

The CLT provides information about the rate at which sample moments converge to population moments as sample size increases.

### 2.3.3 LLN

We begin with the law of large numbers, which tells us when sample averages will converge to their population means.

#### The Classical LLN

The classical law of large numbers concerns independent and identically distributed (IID) random variables.

Here is the strongest version of the classical LLN, known as Kolmogorov's strong law.

Let $X_1, \ldots, X_n$ be independent and identically distributed scalar random variables, with common distribution $F$.

When it exists, let $\mu$ denote the common mean of this sample:

$$\mu := \mathbb{E}X = \int x F(dx)$$

In addition, let

$$\bar{X}_n := \frac{1}{n} \sum_{i=1}^{n} X_i$$

Kolmogorov's strong law states that, if $\mathbb{E}|X|$ is finite, then
\[ P \{ \bar{X}_n \to \mu \text{ as } n \to \infty \} = 1 \quad (2.18) \]

What does this last expression mean?

Let's think about it from a simulation perspective, imagining for a moment that our computer can generate perfect random samples (which of course it can't).

Let's also imagine that we can generate infinite sequences, so that the statement \( \bar{X}_n \to \mu \) can be evaluated.

In this setting, (2.18) should be interpreted as meaning that the probability of the computer producing a sequence where \( \bar{X}_n \to \mu \) fails to occur is zero.

**Proof**

The proof of Kolmogorov's strong law is nontrivial – see, for example, theorem 8.3.5 of [Dud02]. On the other hand, we can prove a weaker version of the LLN very easily and still get most of the intuition.

The version we prove is as follows: If \( X_1, \ldots, X_n \) is IID with \( \mathbb{E}X_i^2 < \infty \), then, for any \( \epsilon > 0 \), we have

\[ P \{ |\bar{X}_n - \mu| \geq \epsilon \} \to 0 \quad \text{as} \quad n \to \infty \quad (2.19) \]

(This version is weaker because we claim only convergence in probability rather than almost sure convergence, and assume a finite second moment.)

To see that this is so, fix \( \epsilon > 0 \), and let \( \sigma^2 \) be the variance of each \( X_i \).

Recall the Chebyshev inequality, which tells us that

\[ P \{ |\bar{X}_n - \mu| \geq \epsilon \} \leq \frac{\mathbb{E}(\bar{X}_n - \mu)^2}{\epsilon^2} \quad (2.20) \]

Now observe that

\[ \mathbb{E}(\bar{X}_n - \mu)^2 = \mathbb{E}\left( \left[ \frac{1}{n} \sum_{i=1}^{n} (X_i - \mu) \right]^2 \right) \]

\[ = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \mathbb{E}(X_i - \mu)(X_j - \mu) \]

\[ = \frac{1}{n^2} \sum_{i=1}^{n} \mathbb{E}(X_i - \mu)^2 \]

\[ = \frac{\sigma^2}{n} \]

Here the crucial step is at the third equality, which follows from independence.

Independence means that if \( i \neq j \), then the covariance term \( \mathbb{E}(X_i - \mu)(X_j - \mu) \) drops out.
As a result, \( n^2 - n \) terms vanish, leading us to a final expression that goes to zero in \( n \). Combining our last result with (2.20), we come to the estimate

\[
\mathbb{P}\left\{ |\bar{X}_n - \mu| \geq \epsilon \right\} \leq \frac{\sigma^2}{n \epsilon^2} \tag{2.21}
\]

The claim in (2.19) is now clear.

Of course, if the sequence \( X_1, \ldots, X_n \) is correlated, then the cross-product terms \( \mathbb{E}(X_i - \mu)(X_j - \mu) \) are not necessarily zero.

While this doesn't mean that the same line of argument is impossible, it does mean that if we want a similar result then the covariances should be almost zero for most of these terms.

In a long sequence, this would be true if, for example, \( \mathbb{E}(X_i - \mu)(X_j - \mu) \) approached zero when the difference between \( i \) and \( j \) became large.

In other words, the LLN can still work if the sequence \( X_1, \ldots, X_n \) has a kind of asymptotic independence, in the sense that correlation falls to zero as variables become further apart in the sequence.

This idea is very important in time series analysis, and we'll come across it again soon enough.

**Illustration**

Let's now illustrate the classical IID law of large numbers using simulation.

In particular, we aim to generate some sequences of IID random variables and plot the evolution of \( \bar{X}_n \) as \( n \) increases.

Below is a figure that does just this (as usual, you can click on it to expand it).

It shows IID observations from three different distributions and plots \( \bar{X}_n \) against \( n \) in each case.

The dots represent the underlying observations \( X_i \) for \( i = 1, \ldots, 100 \).

In each of the three cases, convergence of \( \bar{X}_n \) to \( \mu \) occurs as predicted.
"\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))

num_plots = 3
dist_data = zeros(num_plots, n)
sample_means = []
dist_means = []
titles = []
for i = 1:num_plots
    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 j=1:n
        sample_mean[j] = mean(data[1:j])
    end
    m = mean(dist)
dist_data[i, :] = data'
    push!(sample_means, sample_mean)
    push!(dist_means, m * ones(n))
    push!(titles, name)
end

# == Plot == #
N = repmat(reshape(repmat(1:n, 1, num_plots)', 1, n * num_plots), 2, 1)
heights = [zeros(1, n * num_plots); reshape(dist_data, 1, n * num_plots)]
plot!(1:n, heights, layout=(3, 1), label="", color=:grey, alpha=0.5)
plot!(1:n, dist_data', layout=(3, 1), color=:grey, markershape=:circle,
    alpha=0.5, label="", linewidth=0)
plot!(1:n, sample_means, linewidth=3, alpha=0.6, color=:green, legend=:topleft,
    layout=(3, 1), label=[LaTeXString("$\bar{X}_n\$") "" "")
plot!(1:n, dist_means, color=:black, linewidth=1.5, layout=(3, 1),
    linestyle=:dash, grid=false, label=[LaTeXString("$\mu$") "" "
plot!(title=reshape(titles, 1, length(titles)))
The three distributions are chosen at random from a selection stored in the dictionary `distributions`.

### Infinite Mean

What happens if the condition $E|X| < \infty$ in the statement of the LLN is not satisfied?

This might be the case if the underlying distribution is heavy tailed; the best known example is the Cauchy distribution, which has density

$$f(x) = \frac{1}{\pi(1 + x^2)} \quad (x \in \mathbb{R})$$

The next figure shows 100 independent draws from this distribution.
srand(12)  # reproducible results
n = 200
dist = Cauchy()
data = rand(dist, n)

function plot_draws()
    t = "$n observations from the Cauchy distribution"
    N = repmat(linspace(1, n, n), 1, 2)'
    heights = [zeros(1,n); data']
    plot(1:n, data, color=:blue, markershape=:circle,
         alpha=0.5, title=t, legend=:none, linewidth=0)
    plot!(N, heights, linewidth=0.5, color=:blue)
end

plot_draws()

100 observations from the Cauchy 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

function plot_means()
    # == Compute sample mean at each n == #
    sample_mean = Array{Float64}(n)
    for i=1:n
        sample_mean[i] = mean(data[1:i])
    end

2.3. LLN and CLT
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

\[
\phi(t) = \mathbb{E}e^{itX} = \int e^{itx} f(x) dx = e^{-|t|}
\]  

(2.22)
Using independence, the characteristic function of the sample mean becomes
\[
\mathbb{E} e^{itX_n} = \mathbb{E} \exp \left\{ \frac{t}{n} \sum_{j=1}^{n} X_j \right\} \\
= \mathbb{E} \prod_{j=1}^{n} \exp \left\{ iX_j \right\} \\
= \prod_{j=1}^{n} \mathbb{E} \exp \left\{ iX_j \right\} = [\phi(t/n)]^n
\]

In view of (2.22), 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 \( X_n \) does not converge to a point

### 2.3.4 CLT

Next we turn to the central limit theorem, which tells us about the distribution of the deviation between sample averages and population means

#### Statement of the Theorem

The central limit theorem is one of the most remarkable results in all of mathematics

In the classical IID setting, it tells us the following:

If the sequence \( X_1, \ldots, X_n \) is IID, with common mean \( \mu \) and common variance \( \sigma^2 \in (0, \infty) \), then

\[
\sqrt{n}(\bar{X}_n - \mu) \overset{d}{\to} N(0, \sigma^2) \quad \text{as} \quad n \to \infty \tag{2.23}
\]

Here \( \overset{d}{\to} N(0, \sigma^2) \) 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 $X_1, \ldots, X_n$ be independent. Think of $X_i = 1$ as a success, so that $Y_n = \sum_{i=1}^{n} X_i$ is the number of successes in $n$ trials.

The next figure plots the probability mass function of $Y_n$ for $n = 1, 2, 4, 8$.

```julia
using Distributions

srand(42)  # reproducible results
ns = [1, 2, 4, 8]
dom = 0:9

pdfs = []
titles = []
for n in ns
    b = Binomial(n, 0.5)
    push!(pdfs, pdf(b, dom))
    t = LaTeXString("\$n = \$n\$")
    push!(titles, t)
end

bar(dom, pdfs, layout=4, alpha=0.6, xlims=(-0.5, 8.5), ylims=(0, 0.55),
    xticks=dom, yticks=[0.0, 0.2, 0.4], legend=:none, title=reshape(titles, 1, length(titles)))
```

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)

```julia
# == Set parameters == #
srand(42)  # reproducible results
n = 250    # Choice of n
k = 10000  # 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,..,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 \( Y_n \) == \\
\( Y = \sqrt{n} \ast (\text{sample\_means} - \mu) \)

# == Plot == \\
xmin, xmax = -3 * s, 3 * s \\
\text{histogram}(Y, \text{nbins}=60, \alpha=0.5, \text{xlims}=(\text{xmin, xmax}), \text{norm}=\text{true}, \text{label}="") \\
xgrid = \text{linspace}(\text{xmin, xmax, 200}) \\
\text{plot!}(xgrid, \text{pdf.}(\text{Normal}(0.0, s), xgrid), \text{color}=\text{black}, \text{linewidth}=2, \text{label}=\text{LaTeXString}("\$N(0, \sigma^2=\$(s^2)\$")}, \text{legendfont}=\text{font}(12))

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 \( Y_n := \sqrt{n}(\bar{X}_n - \mu) \) as \( n \) increases

In the simulation well be working with random variables having \( \mu = 0 \)
Thus, when \( n = 1 \), we have \( Y_1 = X_1 \), so the first distribution is just the distribution of the underlying random variable.

For \( n = 2 \), the distribution of \( Y_2 \) is that of \( (X_1 + X_2)/\sqrt{2} \), and so on.

What we expect is that, regardless of the distribution of the underlying random variable, the distribution of \( Y_n \) will smooth out into a bell shaped curve.

The next figure shows this process for \( X_i \sim f \), where \( f \) was specified as the convex combination of three different beta densities.

(Taking a convex combination is an easy way to produce an irregular shape for \( f \)).

In the figure, the closest density is that of \( Y_1 \), while the furthest is that of \( Y_5 \).

```julia
using KernelDensity

beta_dist = Beta(2.0, 2.0)

function gen_x_draws(k)
    bdraws = rand(beta_dist, 3, k)

    # == Transform rows, so each represents a different distribution ==#
    bdraws[1, :] -= 0.5
    bdraws[2, :] += 0.6
    bdraws[3, :] -= 1.1

    # == Set \( X[i] = bdraws[j, i] \), where \( j \) is a random draw from \{1, 2, 3\} ==#
    js = rand(1:3, k)
    X = Array(Float64)(k)
    for i=1:k
        X[i] = bdraws[js[i], i]
    end

    # == Rescale, so that the random variable is zero mean ==#
    m, sigma = mean(X), std(X)
    return (X .- m) ./ sigma
end

nmax = 5
reps = 100000
ns = 1:nmax

# == Form a matrix \( Z \) such that each column is \( reps \) independent draws of \( X \) ==#
Z = Array(Float64)(reps, nmax)
for i=ns
    Z[:, i] = gen_x_draws(reps)
end

# == Take cumulative sum across columns
S = cumsum(Z, 2)

# == Multiply \( j \)-th column by \( \sqrt{j} \) ==#
Y = S .* (1. ./ sqrt.(ns))'
```

2.3. LLN and CLT
As expected, the distribution smooths out into a bell curve as \( n \) increases.

We leave you to investigate its contents if you wish to know more.
If you run the file from the ordinary 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 \( X \) is just a sequence of \( k \) random variables \((X_1, \ldots, X_k)\).

Each realization of \( X \) is an element of \( \mathbb{R}^k \).

A collection of random vectors \( X_1, \ldots, X_n \) is called independent if, given any \( n \) vectors \( x_1, \ldots, x_n \) in \( \mathbb{R}^k \), we have

\[
P\{X_1 \leq x_1, \ldots, X_n \leq x_n\} = P\{X_1 \leq x_1\} \times \cdots \times P\{X_n \leq x_n\}
\]

(The vector inequality \( X \leq x \) means that \( X_j \leq x_j \) for \( j = 1, \ldots, k \)).

Let \( \mu_j := E[X_j] \) for all \( j = 1, \ldots, k \).

The expectation \( E[X] \) of \( X \) is defined to be the vector of expectations:

\[
E[X] := \begin{pmatrix} E[X_1] \\ E[X_2] \\ \vdots \\ E[X_k] \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_k \end{pmatrix} =: \mu
\]

The variance-covariance matrix of random vector \( X \) is defined as

\[
Var[X] := E[(X - \mu)(X - \mu)']
\]

Expanding this out, we get

\[
Var[X] = \begin{pmatrix} E[(X_1 - \mu_1)(X_1 - \mu_1)] & \cdots & E[(X_1 - \mu_1)(X_k - \mu_k)] \\ E[(X_2 - \mu_2)(X_1 - \mu_1)] & \cdots & E[(X_2 - \mu_2)(X_k - \mu_k)] \\ \vdots & \ddots & \vdots \\ E[(X_k - \mu_k)(X_1 - \mu_1)] & \cdots & E[(X_k - \mu_k)(X_k - \mu_k)] \end{pmatrix}
\]

The \( j, k \)-th term is the scalar covariance between \( X_j \) and \( X_k \).

With this notation we can proceed to the multivariate LLN and CLT.

Let \( X_1, \ldots, X_n \) be a sequence of independent and identically distributed random vectors, each one taking values in \( \mathbb{R}^k \).

Let \( \mu \) be the vector \( E[X_i] \), and let \( \Sigma \) be the variance-covariance matrix of \( X_i \).

Interpreting vector addition and scalar multiplication in the usual way (i.e., pointwise), let

\[
X_n := \frac{1}{n} \sum_{i=1}^n X_i
\]
In this setting, the LLN tells us that

\[
P \left\{ \bar{X}_n \to \mu \mbox{ as } n \to \infty \right\} = 1 \tag{2.24}
\]

Here \( \bar{X}_n \to \mu \) means that \( \| \bar{X}_n - \mu \| \to 0 \), where \( \| \cdot \| \) is the standard Euclidean norm.

The CLT tells us that, provided \( \Sigma \) is finite,

\[
\sqrt{n}(\bar{X}_n - \mu) \overset{d}{\to} N(0, \Sigma) \quad \text{as } n \to \infty \tag{2.25}
\]

### 2.3.5 Exercises

**Exercise 1**

One very useful consequence of the central limit theorem is as follows.

Assume the conditions of the CLT as stated above.

If \( g: \mathbb{R} \to \mathbb{R} \) is differentiable at \( \mu \) and \( g'(\mu) \neq 0 \), then

\[
\sqrt{n}\{g(\bar{X}_n) - g(\mu)\} \overset{d}{\to} N(0, g'(\mu)^2 \sigma^2) \quad \text{as } n \to \infty \tag{2.26}
\]

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. \( X_1, \ldots, X_n \) is a sequence of IID random vectors, each taking values in \( \mathbb{R}^k \).
2. \( \mu := \mathbb{E}[X_i] \), and \( \Sigma \) is the variance-covariance matrix of \( X_i \).

3. The convergence

\[
\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} N(0, \Sigma)
\]

(2.27)

is valid

In a statistical setting, one often wants the right hand side to be standard normal, so that confidence intervals are easily computed.

This normalization can be achieved on the basis of three observations:

First, if \( X \) is a random vector in \( \mathbb{R}^k \) and \( A \) is constant and \( k \times k \), then

\[
\text{Var}[AX] = A \text{Var}[X]A'
\]

Second, by the continuous mapping theorem, if \( Z_n \xrightarrow{d} Z \) in \( \mathbb{R}^k \) and \( A \) is constant and \( k \times k \), then

\[
AZ_n \xrightarrow{d} AZ
\]

Third, if \( S \) is a \( k \times k \) symmetric positive definite matrix, then there exists a symmetric positive definite matrix \( Q \), called the inverse square root of \( S \), such that

\[
QSQ' = I
\]

Here \( I \) is the \( k \times k \) identity matrix.

Putting these things together, your first exercise is to show that if \( Q \) is the inverse square root of \( \Sigma \), then

\[
Z_n := \sqrt{n}Q(\bar{X}_n - \mu) \xrightarrow{d} Z \sim N(0, I)
\]

Applying the continuous mapping theorem one more time tells us that

\[
\|Z_n\|^2 \xrightarrow{d} \|Z\|^2
\]

Given the distribution of \( Z \), we conclude that

\[
n\|Q(\bar{X}_n - \mu)\|^2 \xrightarrow{d} \chi^2(k)
\]

(2.28)

where \( \chi^2(k) \) is the chi-squared distribution with \( k \) degrees of freedom.

(Recall that \( k \) is the dimension of \( X_i \), the underlying random vectors.)

Your second exercise is to illustrate the convergence in (2.28) with a simulation.

In doing so, let

\[
X_i := \begin{pmatrix} W_i \\ U_i + W_i \end{pmatrix}
\]

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. $\text{sqrtm}(A)$ computes the square root of $A$. You still need to invert it
2. You should be able to work out $\Sigma$ from the proceeding information

2.3.6 Solutions

Exercise 1

Here is one solution

You might have to modify or delete the lines starting with `rc`, depending on your configuration

```julia
# == Set parameters == #
srand(42)  # reproducible results
n = 250    # Choice of n
k = 100000 # Number of draws of $Y_n$
dist = Uniform(0, π/2)
μ, s = mean(dist), std(dist)
g = sin
g = cos

# == Draw underlying RVs. Each row contains a draw of $X_1, \ldots, X_n == #
data = rand(dist, k, n)

# == Compute mean of each row, producing $k$ draws of $\bar{X}_n == #
sample_means = mean(data, 2)

error_obs = sqrt(n) .* (g.(sample_means) - g.(μ))

# == Plot == #
asymptotic_sd = g(μ) .* s
xmin = -3 * g(μ) * s
xmax = -xmin
histogram(error_obs, nbins=60, alpha=0.5, normed=true, label="")
xgrid = linspace(xmin, xmax, 200)
plot!(xgrid, pdf.(Normal(0.0, asymptotic_sd), xgrid), color=:black,
      linewidth=2, label=LaTeXString("\(N(0, g'(\mu)^2\sigma^2)\)"),
      legendfont=font(12), xlims=(xmin, xmax), grid=false)
```
What happens when you replace \([0, \pi/2]\) with \([0, \pi]\)?

In this case, the mean \(\mu\) of this distribution is \(\pi/2\), and since \(g' = \cos\), we have \(g'(\mu) = 0\).

Hence the conditions of the delta theorem are not satisfied.

**Exercise 2**

First we want to verify the claim that

\[ \sqrt{n}Q(X_n - \mu) \xrightarrow{d} N(0, I) \]

This is straightforward given the facts presented in the exercise.

Let

\[ Y_n := \sqrt{n}(\bar{X}_n - \mu) \quad \text{and} \quad Y \sim N(0, \Sigma) \]

By the multivariate CLT and the continuous mapping theorem, we have

\[ QY_n \xrightarrow{d} QY \]

Since linear combinations of normal random variables are normal, the vector \(QY\) is also normal. Its mean is clearly \(0\), and its variance covariance matrix is

\[ \text{Var}[QY] = Q\text{Var}[Y]Q' = Q\Sigma Q' = I \]
In conclusion, \( Q Y_n \xrightarrow{d} Q Y \sim N(0, I) \), which is what we aimed to show.

Now we turn to the simulation exercise.

Our solution is as follows

```julia
# == Set parameters == #
\n\text{n} = 250
\text{replications} = 50000
\text{dw} = \text{Uniform}(\text{-1, 1})
\text{du} = \text{Uniform}(\text{-2, 2})
\text{sw, su} = \text{std}(\text{dw}), \text{std}(\text{du})
\text{vw, vu} = \text{sw}^2, \text{su}^2
\Sigma = \begin{bmatrix} \text{vw} & \text{vw} \\ \text{vw} & \text{vw} + \text{vu} \end{bmatrix}

# == Compute \( \Sigma^{\frac{-1}{2}} \) == #
Q = \text{inv} (\sqrtm (\Sigma))

# == Generate observations of the normalized sample mean == #
error_obs = \text{Array} (\text{Float64}) (2, \text{replications})
\text{for}\ i = 1: \text{replications}
    # == Generate one sequence of bivariate shocks == #
    X = \text{Array} (\text{Float64}) (2, \text{n})
    W = \text{rand} (\text{dw}, \text{n})
    U = \text{rand} (\text{du}, \text{n})

    # == Construct the \text{n} observations of the random vector == #
    X[1, :] = W
    X[2, :] = W + U

    # == Construct the \text{i}-th observation of \text{Y_n} == #
    error_obs[:, i] = \text{sqrt (n)} .* \text{mean} (X, 2)
\text{end}

chisq_obs = \text{squeeze} (\text{sum} ((Q * \text{error_obs}).^2, 1), 1)

# == Plot == #
xmin, xmax = 0, 8
\text{histogram} (\text{chisq_obs}, \text{nbins}=50, \text{normed}=\text{true}, \text{label}="")
xgrid = \text{linspace} (\text{xmin}, \text{xmax}, 200)
\text{plot!} (\text{xgrid}, \text{pdf} (\text{Chisq}(2), \text{xgrid}), \text{color}=\text{black},
\quad \text{linewidth}=2, \text{label}="\text{Chi-squared with 2 degrees of freedom}",
\quad \text{legendfont}=\text{font} (12), \text{xlims}=(\text{xmin}, \text{xmax}), \text{grid}=\text{false})
```

202 Chapter 2. Tools and Techniques
2.4 Linear State Space Models

Content

- 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
2.4.1 Overview

This lecture introduces the linear state space dynamic system

This model is a workhorse that carries a powerful theory of prediction

Its many applications include:

• representing dynamics of higher-order linear systems
• predicting the position of a system \( j \) steps into the future
• predicting a geometric sum of future values of a variable like
  – non financial income
  – dividends on a stock
  – the money supply
  – a government deficit or surplus, etc.
• key ingredient of useful models
  – Friedmans permanent income model of consumption smoothing
  – Barros model of smoothing total tax collections
  – Rational expectations version of Cagans model of hyperinflation
  – Sargent and Wallaces unpleasant monetarist arithmetic, etc.

2.4.2 The Linear State Space Model

The objects in play are:

• An \( n \times 1 \) vector \( x_t \) denoting the state at time \( t = 0, 1, 2, \ldots \)
• An iid sequence of \( m \times 1 \) random vectors \( w_t \sim N(0, I) \)
• A \( k \times 1 \) vector \( y_t \) of observations at time \( t = 0, 1, 2, \ldots \)
• An \( n \times n \) matrix \( A \) called the transition matrix
• An \( n \times m \) matrix \( C \) called the volatility matrix
• A \( k \times n \) matrix \( G \) sometimes called the output matrix

Here is the linear state-space system

\[
\begin{align*}
x_{t+1} &= Ax_t + Cw_{t+1} \\
y_t &= Gx_t \\
x_0 &\sim N(\mu_0, \Sigma_0)
\end{align*}
\]
Primitives

The primitives of the model are

1. the matrices $A, C, G$
2. shock distribution, which we have specialized to $N(0, I)$
3. the distribution of the initial condition $x_0$, which we have set to $N(\mu_0, \Sigma_0)$

Given $A, C, G$ and draws of $x_0$ and $w_1, w_2, \ldots$, the model (2.29) pins down the values of the sequences $\{x_t\}$ and $\{y_t\}$

Even without these draws, the primitives 1–3 pin down the probability distributions of $\{x_t\}$ and $\{y_t\}$

Later well see how to compute these distributions and their moments

Martingale difference shocks

Weve made the common assumption that the shocks are independent standardized normal vectors

But some of what we say will be valid under the assumption that $\{w_{t+1}\}$ is a martingale difference sequence

A martingale difference sequence is a sequence that is zero mean when conditioned on past information

In the present case, since $\{x_t\}$ is our state sequence, this means that it satisfies

$$E[w_{t+1}|x_t, x_{t-1}, \ldots] = 0$$

This is a weaker condition than that $\{w_t\}$ is iid with $w_{t+1} \sim N(0, I)$

Examples

By appropriate choice of the primitives, a variety of dynamics can be represented in terms of the linear state space model

The following examples help to highlight this point

They also illustrate the wise dictum finding the state is an art

Second-order difference equation

Let $\{y_t\}$ be a deterministic sequence that satisfies

$$y_{t+1} = \phi_0 + \phi_1 y_t + \phi_2 y_{t-1} \quad \text{s.t.} \quad y_0, y_{-1} \text{ given} \quad (2.29)$$

To map (2.29) into our state space system (2.29), we set

$$x_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \phi_0 & \phi_1 & \phi_2 \\ 0 & 1 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad G = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$
You can confirm that under these definitions, (2.29) 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 (2.29) to represent the model

$$y_{t+1} = \phi_1 y_t + \phi_2 y_{t-1} + \phi_3 y_{t-2} + \phi_4 y_{t-3} + \sigma w_{t+1}$$  \hspace{1cm} (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

$$A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

The matrix $A$ has the form of the \textit{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 (2.29), we set

$$x_t = \begin{bmatrix} y_t \\ y_{t-1} \\ y_{t-2} \\ y_{t-3} \end{bmatrix}, \quad A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \end{bmatrix}, \quad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad G = \begin{bmatrix} I & 0 & 0 \end{bmatrix}$$

where $I$ is the $k \times k$ identity matrix and $\sigma$ is a $k \times k$ matrix

Seasonals

We can use (2.29) 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:

$$x_{t+4} = x_t$$

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

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad G = [a \ b]$$

and starting at initial condition $x_0 = [0 \ 1]'$

In fact its possible to use the state-space system to represent polynomial trends of any order

For instance, let

$$x_0 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

It follows that

$$A^t = \begin{bmatrix} 1 & t & t(t-1)/2 \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix}$$

Then $x'_t = [t(t-1)/2 \ t \ 1]$, so that $x_t$ contains linear and quadratic time trends

\footnote{The eigenvalues of $A$ are $(1, -1, i, -i)$.}
Moving Average Representations

A nonrecursive expression for $x_t$ as a function of $x_0, w_1, w_2, \ldots, w_t$ can be found by using (2.29) repeatedly to obtain

$$
x_t = Ax_{t-1} + Cw_t
$$

$$
= A^2x_{t-2} + ACw_{t-1} + Cw_t
$$

$$
\vdots
$$

$$
= \sum_{j=0}^{t-1} A^jCw_{t-j} + A^tx_0
$$

Representation (2.32) is a moving average representation.

It expresses $\{x_t\}$ as a linear function of

1. current and past values of the process $\{w_t\}$
2. the initial condition $x_0$

As an example of a moving average representation, let the model be

$$
A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 \\ 0 \end{bmatrix}
$$

You will be able to show that $A^t = \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix}$ and $A^tC = \begin{bmatrix} 1 & 0 \end{bmatrix}'$

Substituting into the moving average representation (2.32), we obtain

$$
x_{1t} = \sum_{j=0}^{t-1} w_{t-j} + \begin{bmatrix} 1 & t \end{bmatrix} x_0
$$

where $x_{1t}$ is the first entry of $x_t$

The first term on the right is a cumulated sum of martingale differences, and is therefore a martingale

The second term is a translated linear function of time.

For this reason, $x_{1t}$ is called a martingale with drift.

2.4.3 Distributions and Moments

Unconditional Moments

Using (2.29), it’s easy to obtain expressions for the (unconditional) means of $x_t$ and $y_t$

Well explain what unconditional and conditional mean soon.

Letting $\mu_t := \mathbb{E}[x_t]$ and using linearity of expectations, we find that
\[ \mu_{t+1} = A \mu_t \quad \text{with} \quad \mu_0 \text{ given} \]  

(2.32)

Here \( \mu_0 \) is a primitive given in (2.29)

The variance-covariance matrix of \( x_t \) is \( \Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)'] \)

Using \( x_{t+1} - \mu_{t+1} = A(x_t - \mu_t) + C w_{t+1} \), we can determine this matrix recursively via

\[ \Sigma_{t+1} = A \Sigma_t A' + CC' \quad \text{with} \quad \Sigma_0 \text{ given} \]  

(2.33)

As with \( \mu_0 \), the matrix \( \Sigma_0 \) is a primitive given in (2.29)

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

\[ \mathbb{E}[y_t] = \mathbb{E}[G x_t] = G \mu_t \]  

(2.34)

The variance-covariance matrix of \( y_t \) is easily shown to be

\[ \text{Var}[y_t] = \text{Var}[G x_t] = G \Sigma_t G' \]  

(2.35)

**Distributions**

In general, knowing the mean and variance-covariance matrix of a random vector is not quite as good as knowing the full distribution

However, there are some situations where these moments alone tell us all we need to know

These are situations in which the mean vector and covariance matrix are sufficient statistics for the population distribution

(Sufficient statistics form a list of objects that characterize a population distribution)

One such situation is when the vector in question is Gaussian (i.e., normally distributed)

This is the case here, given
1. our Gaussian assumptions on the primitives
2. the fact that normality is preserved under linear operations

In fact, it’s well-known that

\[ u \sim N(\bar{u}, S) \quad \text{and} \quad v = a + Bu \implies v \sim N(a + B\bar{u}, BSB') \] (2.36)

In particular, given our Gaussian assumptions on the primitives and the linearity of (2.29) we can see immediately that both \( x_t \) and \( y_t \) are Gaussian for all \( t \geq 0 \).

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 (2.32) and (2.33)

Letting \( \mu_t \) and \( \Sigma_t \) be as defined by these equations, we have

\[ x_t \sim N(\mu_t, \Sigma_t) \] (2.37)

By similar reasoning combined with (2.34) and (2.35),

\[ y_t \sim N(G\mu_t, G\Sigma_t G') \] (2.38)

**Ensemble Interpretations**

How should we interpret the distributions defined by (2.37)–(2.38)?

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.

---

2 The correct way to argue this is by induction. Suppose that \( x_t \) is Gaussian. Then (2.29) and (2.36) 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.
In the right-hand figure, these values are converted into a rotated histogram that shows relative frequencies from our sample of 20 \( y_T \)s.

(The parameters and source code for the figures can be found in file `linear_models/paths_and_hist.jl`)

Here is another figure, this time with 100 observations

Let's now try with 500,000 observations, showing only the histogram (without rotation)
The black line is the population density of $y_T$ calculated from (2.38)

The histogram and population distribution are close, as expected

By looking at the figures and experimenting with parameters, you will gain a feel for how the population distribution depends on the model primitives listed above, as intermediated by the distributions sufficient statistics

**Ensemble means**

In the preceding figure we approximated the population distribution of $y_T$ by

1. generating $I$ sample paths (i.e., time series) where $I$ is a large number
2. recording each observation $y^i_T$
3. histogramming this sample

Just as the histogram approximates the population distribution, the *ensemble or cross-sectional average*

$$
\bar{y}_T := \frac{1}{I} \sum_{i=1}^{I} y^i_T
$$

approximates the expectation $E[y_T] = G \mu_T$ (as implied by the law of large numbers)

Here’s a simulation comparing the ensemble averages and population means at time points $t = 0, \ldots, 50$

The parameters are the same as for the preceding figures, and the sample size is relatively small ($I = 20$)
The ensemble mean for $x_t$ is

$$\bar{x}_T := \frac{1}{I} \sum_{i=1}^{I} x_T^i \rightarrow \mu_T \quad (I \rightarrow \infty)$$

The limit $\mu_T$ is a long-run average

(By long-run average we mean the average for an infinite ($I = \infty$) number of sample $x_T$s)

Another application of the law of large numbers assures us that

$$\frac{1}{I} \sum_{i=1}^{I} (x_T^i - \bar{x}_T)(x_T^i - \bar{x}_T)^{\prime} \rightarrow \Sigma_T \quad (I \rightarrow \infty)$$

**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 \( x_0, x_1, \ldots, x_T \), recall that joint and conditional densities are linked by the rule

\[
p(x, y) = p(y | x)p(x) \quad \text{(joint = conditional \times marginal)}
\]

From this rule we get \( p(x_0, x_1) = p(x_1 | x_0)p(x_0) \)

The Markov property \( p(x_t | x_{t-1}, \ldots, x_0) = p(x_t | x_{t-1}) \) and repeated applications of the preceding rule lead us to

\[
p(x_0, x_1, \ldots, x_T) = p(x_0) \prod_{t=0}^{T-1} p(x_{t+1} | x_t)
\]

The marginal \( p(x_0) \) is just the primitive \( N(\mu_0, \Sigma_0) \)

In view of (2.29), 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*

\[
\Sigma_{t+j,t} := \mathbb{E}[(x_{t+j} - \mu_{t+j})(x_t - \mu_t)']
\]  
(2.39)

Elementary calculations show that

\[
\Sigma_{t+j,t} = A^j \Sigma_t
\]  
(2.40)

Notice that \( \Sigma_{t+j,t} \) in general depends on both \( j \), the gap between the two dates, and \( t \), the earlier date

### 2.4.4 Stationarity and Ergodicity

Stationarity and ergodicity are two properties that, when they hold, greatly aid analysis of linear state space models

Let’s start with the intuition

**Visualizing Stability**

Let’s look at some more time series from the same model that we analyzed above

This picture shows cross-sectional distributions for \( y \) at times \( T, T', T'' \)
Note how the time series settle down in the sense that the distributions at $T'$ and $T''$ are relatively similar to each other but unlike the distribution at $T$.

Apparently, the distributions of $y_t$ converge to a fixed long-run distribution as $t \to \infty$.

When such a distribution exists it is called a *stationary distribution*.

**Stationary Distributions**

In our setting, a distribution $\psi_{\infty}$ is said to be *stationary* for $x_t$ if

$$x_t \sim \psi_{\infty} \quad \text{and} \quad x_{t+1} = Ax_t + Cw_{t+1} \quad \Rightarrow \quad x_{t+1} \sim \psi_{\infty}$$

Since

1. in the present case all distributions are Gaussian
2. a Gaussian distribution is pinned down by its mean and variance-covariance matrix

we can restate the definition as follows: $\psi_{\infty}$ is stationary for $x_t$ if

$$\psi_{\infty} = N(\mu_{\infty}, \Sigma_{\infty})$$

where $\mu_{\infty}$ and $\Sigma_{\infty}$ are fixed points of (2.32) and (2.33) 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 (2.32) and (2.33) respectively

we've ensured that

$$\mu_t = \mu_\infty \quad \text{and} \quad \Sigma_t = \Sigma_\infty \quad \text{for all} \quad t$$

Moreover, in view of (2.40), 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 $t$.

### 2.4. Linear State Space Models
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 (2.33) also has a unique fixed point in this case, and, moreover

\[ \mu_t \to \mu_\infty = 0 \quad \text{and} \quad \Sigma_t \to \Sigma_\infty \quad \text{as} \quad t \to \infty \]

regardless of the initial conditions $\mu_0$ and $\Sigma_0$.

This is the globally stable case; see these notes for more 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 $\mu_t \to 0$.

How can we find stationary solutions that respect a constant state component?

Processes with a constant state component

To investigate such a process, suppose that $A$ and $C$ take the form

\[ A = \begin{bmatrix} A_1 & a \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} C_1 \\ 0 \end{bmatrix} \]

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

\[ \mu_{1,t+1} = A_1 \mu_{1t} + a \quad (2.41) \]

Assume now that the moduli of the eigenvalues of $A_1$ are all strictly less than one.
Then (2.41) has a unique stationary solution, namely,

$$\mu_{1\infty} = (I - A_1)^{-1}a$$

The stationary value of $\mu_t$ itself is then $\mu_{\infty} := [\mu_{1\infty}' \quad 1]'$

The stationary values of $\Sigma_t$ and $\Sigma_{t+j,t}$ satisfy

$$\Sigma_{\infty} = A\Sigma_{\infty} A' + C C'$$
$$\Sigma_{t+j,t} = A^j \Sigma_{\infty}$$

Notice that here $\Sigma_{t+j,t}$ depends on the time gap $j$ but not on calendar time $t$

In conclusion, if

- $x_0 \sim N(\mu_{\infty}, \Sigma_{\infty})$ and
- the moduli of the eigenvalues of $A_1$ are all strictly less than unity

then the $\{x_t\}$ process is covariance stationary, with constant state component

**Note:** If the eigenvalues of $A_1$ are less than unity in modulus, then (a) starting from any initial value, the mean and variance-covariance matrix both converge to their stationary values; and (b) iterations on (2.33) converge to the fixed point of the discrete Lyapunov equation in the first line of (2.42)

**Ergodicity**

Lets suppose that we are 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

---

2.4. Linear State Space Models
More formally, ergodicity implies that time series sample averages converge to their expectation under the stationary distribution.

In particular,

- \( \frac{1}{T} \sum_{t=1}^{T} x_t \rightarrow \mu_\infty \)
- \( \frac{1}{T} \sum_{t=1}^{T} (x_t - \bar{x}_T)(x_t - \bar{x}_T)' \rightarrow \Sigma_\infty \)
- \( \frac{1}{T} \sum_{t=1}^{T} (x_{t+j} - \bar{x}_T)(x_t - \bar{x}_T)' \rightarrow A^j \Sigma_\infty \)

In our linear Gaussian setting, any covariance stationary process is also ergodic.

### 2.4.5 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

\[
\begin{align*}
    &x_{t+1} = Ax_t + Cw_{t+1} \\
    &y_t = Gx_t + Hv_t \\
    &x_0 \sim N(\mu_0, \Sigma_0)
\end{align*}
\]

The sequence \( \{v_t\} \) is assumed to be independent of \( \{w_t\} \).

The process \( \{x_t\} \) is not modified by noise in the observation equation and its moments, distributions and stability properties remain the same.

The unconditional moments of \( y_t \) from (2.34) and (2.35) now become

\[
E[y_t] = E[Gx_t + Hv_t] = G\mu_t
\]

The variance-covariance matrix of \( y_t \) is easily shown to be

\[
\text{Var}[y_t] = \text{Var}[Gx_t + Hv_t] = G\Sigma_t G' + HH'
\]

The distribution of \( y_t \) is therefore

\[
y_t \sim N(G\mu_t, G\Sigma_t G' + HH')
\]
2.4.6 Prediction

The theory of prediction for linear state space systems is elegant and simple.

**Forecasting Formulas – Conditional Means**

The natural way to predict variables is to use conditional distributions. For example, the optimal forecast of \( x_{t+1} \) given information known at time \( t \) is

\[
E_t[x_{t+1}] := E[x_{t+1} \mid x_t, x_{t-1}, \ldots, x_0] = Ax_t
\]

The right-hand side follows from \( x_{t+1} = Ax_t + Cw_{t+1} \) and the fact that \( w_{t+1} \) is zero mean and independent of \( x_t, x_{t-1}, \ldots, x_0 \).

That \( E_t[x_{t+1}] = E[x_{t+1} \mid x_t] \) is an implication of \( \{x_t\} \) having the *Markov property*.

The one-step-ahead forecast error is

\[
x_{t+1} - E_t[x_{t+1}] = Cw_{t+1}
\]

The covariance matrix of the forecast error is

\[
E[(x_{t+1} - E_t[x_{t+1}])(x_{t+1} - E_t[x_{t+1}])'] = CC'
\]

More generally, we'd like to compute the \( j \)-step ahead forecasts \( E_t[x_{t+j}] \) and \( E_t[y_{t+j}] \).

With a bit of algebra we obtain

\[
x_{t+j} = A^j x_t + A^{j-1}Cw_{t+1} + A^{j-2}Cw_{t+2} + \cdots + A^0Cw_{t+j}
\]

In view of the iid property, current and past state values provide no information about future values of the shock.

Hence \( E_t[w_{t+k}] = E[w_{t+k}] = 0 \).

It now follows from linearity of expectations that the \( j \)-step ahead forecast of \( x \) is

\[
E_t[x_{t+j}] = A^j x_t
\]

The \( j \)-step ahead forecast of \( y \) is therefore

\[
E_t[y_{t+j}] = E_t[Gx_{t+j} + Hv_{t+j}] = GA^j x_t
\]

**Covariance of Prediction Errors**

It is useful to obtain the covariance matrix of the vector of \( j \)-step-ahead prediction errors

\[
x_{t+j} - E_t[x_{t+j}] = \sum_{s=0}^{j-1} A^s Cw_{t-s+j}
\]

2.4. Linear State Space Models
Evidently,

\[ V_j := \mathbb{E}_t[(x_{t+j} - \mathbb{E}_t[x_{t+j}])(x_{t+j} - \mathbb{E}_t[x_{t+j}])'] = \sum_{k=0}^{j-1} A^k CC' A^{k'} \]  

(2.45)

\( V_j \) defined in (2.45) can be calculated recursively via

\[ V_1 = CC' \] and

\[ V_j = CC' + AV_{j-1}A', \quad j \geq 2 \]  

(2.46)

\( V_j \) 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

\[ V_\infty = CC' + AV_\infty A' \]  

(2.47)

Equation (2.47) 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 (2.29)

We want the following objects

- Forecast of a geometric sum of future \( x \)s, or \( \mathbb{E}_t \left[ \sum_{j=0}^{\infty} \beta^j x_{t+j} \right] \)

- Forecast of a geometric sum of future \( y \)s, or \( \mathbb{E}_t \left[ \sum_{j=0}^{\infty} \beta^j y_{t+j} \right] \)

These objects are important components of some famous and interesting dynamic models

For example,

- If \( \{ y_t \} \) is a stream of dividends, then \( \mathbb{E} \left[ \sum_{j=0}^{\infty} \beta^j y_{t+j} | x_t \right] \) is a model of a stock price

- If \( \{ y_t \} \) is the money supply, then \( \mathbb{E} \left[ \sum_{j=0}^{\infty} \beta^j y_{t+j} | x_t \right] \) is a model of the price level
**Formulas**

Fortunately, it is easy to use a little matrix algebra to compute these objects.

Suppose that every eigenvalue of \( A \) has modulus strictly less than \( \frac{1}{\beta} \).

It *then follows* that

\[
I + \beta A + \beta^2 A^2 + \cdots = [I - \beta A]^{-1}
\]

This leads to our formulas:

- **Forecast of a geometric sum of future \( x \)s**
  
  \[
  \mathbb{E}_t \left[ \sum_{j=0}^{\infty} \beta^j x_{t+j} \right] = [I + \beta A + \beta^2 A^2 + \cdots] x_t = [I - \beta A]^{-1} x_t
  \]

- **Forecast of a geometric sum of future \( y \)s**
  
  \[
  \mathbb{E}_t \left[ \sum_{j=0}^{\infty} \beta^j y_{t+j} \right] = G[I + \beta A + \beta^2 A^2 + \cdots] x_t = G[I - \beta A]^{-1} x_t
  \]

**2.4.7 Code**

Our preceding simulations and calculations are based on code in the file `lss.jl` from the `QuantEcon.jl` package.

The code implements a type which the linear state space models can act on directly through specific methods (for simulations, calculating moments, etc.).

Examples of usage are given in the solutions to the exercises.

**2.4.8 Exercises**

**Exercise 1**

Replicate *this figure* using the `LSS` 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

2.4.9 Solutions

```julia
using QuantEcon
using Plots
pyplot()
using LaTeXStrings

Exercise 1

₀,₁,₂ = 1.1, 0.8, -0.8
A = [1.0 0.0 0
     ₀ ₁ ₂
     0.0 1.0 0.0]
C = zeros(3, 1)
G = [0.0 1.0 0.0]
μ₀ = ones(3)
lss = LSS(A, C, G; μ₀=μ₀)
x, y = simulate(lss, 50)
plot(squeeze(y, 1), color=:blue, linewidth=2, alpha=0.7)
plot!(xlabel="time", ylabel="yₜ", legend=:none)
```
Exercise 2

\[ _1, _2, _3, _4 = 0.5, -0.2, 0, 0.5 \]
\[ \sigma = 0.2 \]
\[ A = \begin{bmatrix} _1 & _2 & _3 & _4 \\ 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \end{bmatrix} \]
\[ C = \begin{bmatrix} \sigma \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} \]
\[ G = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \]

ar = LSS(A, C, G; mu_0=ones(4))
x, y = simulate(ar, 200)

plot(squeeze(y, 1), color=:blue, linewidth=2, alpha=0.7)
plot!(xlabel="time", ylabel=L"$y_t$", legend=:none)
Exercise 3

\[ _1, _2, _3, _4 = 0.5, -0.2, 0, 0.5 \]
\[ \sigma = 0.1 \]

\[ A = \begin{bmatrix} _1 & _2 & _3 & _4 \end{bmatrix} \]
\[ = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 1.0 & 0.0 \end{bmatrix} \]

\[ C = [\sigma] \]
\[ = [0.0] \]

\[ G = [1.0, 0.0, 0.0, 0.0] \]
\[ I = 20 \]
\[ T = 50 \]
\[ ar = \text{LSS}(A, C, G; \text{mu}_0=\text{ones}(4)) \]
\[ \text{ymin, ymax} = -0.5, 1.15 \]

\[ \text{ensemble\_mean} = \text{zeros}(T) \]
\[ \text{ys} = [] \]
\[ \text{for } i=1:1 \]
\[ \quad x, y \text{ = simulate}(ar, T) \]
\[ \quad y = \text{squeeze}(y, 1) \]
\[ \quad \text{push!}(\text{ys}, y) \]
\[ \quad \text{ensemble\_mean .+= y} \]
\[ \text{end} \]
ensemble_mean = ensemble_mean ./ I
plot(ys, color=:blue, alpha=0.2, linewidth=0.8, label=""
plot!(ensemble_mean, color=:blue, linewdith=2, label=L"\$\text{bar y}_t\$")
m = moment_sequence(ar)
state = start(m)
pop_means = Float64[
for t=1:T
    (μ_x, μ_y, Σ_x, Σ_y), state = next(m, state)
    push!(pop_means, μ_y[1])
end
plot!(pop_means, color=:green, linewidth=2, label=L"G\mu_t")
plot!(ylims=(ymin, ymax), xlabel="time", ylabel="y_t", legendfont=font(12))
G = [1.0 0.0 0.0 0.0]
T0 = 10
T1 = 50
T2 = 75
T4 = 100
ar = LSS(A, C, G; mu_0=ones(4))
ymin, ymax = -0.6, 0.6
μ_x, μ_y, Σ_x, Σ_y = stationary_distributions(ar)
ar = LSS(A, C, G; mu_0=μ_x, Sigma_0=Σ_x)
colors = ["c", "g", "b"]
ys = []
_x_scatter = []
_y_scatter = []
for i=1:80
    rcolor = colors[rand(1:3)]
x, y = simulate(ar, T4)
y = squeeze(y, 1)
push!(ys, y)
    _x_scatter = [_x_scatter; T0; T1; T2]
    _y_scatter = [_y_scatter; y[T0]; y[T1]; y[T2]]
end
plot(ys, linewidth=0.8, alpha=0.5)
plot!(_x_scatter, _y_scatter, color=:black, legend=:none)
scatter!(ys, linewidth=0.8, alpha=0.5)
plot!(ylims=(ymin, ymax), ylabel="\$y_t\$", xticks=[], yticks=ymin:0.2:ymax)
plot!(annotations=[(T0+1, -0.55, L"$T_0$");(T1+1, -0.55, L"$T_1$");(T2+1, -0.55, L"$T_2$"))]
2.5 Finite Markov Chains

Contents

• Finite Markov Chains
  – Overview
  – Definitions
  – Simulation
  – Marginal Distributions
  – Irreducibility and Aperiodicity
  – Stationary Distributions
  – Ergodicity
  – Computing Expectations
  – Exercises
  – Solutions
2.5.1 Overview

Markov chains are one of the most useful classes of stochastic processes, being

- simple, flexible and supported by many elegant theoretical results
- valuable for building intuition about random dynamic models
- central to quantitative modeling in their own right

You will find them in many of the workhorse models of economics and finance.

In this lecture we review some of the theory of Markov chains.

We will also introduce some of the high quality routines for working with Markov chains available in 
QuantEcon.jl

Prerequisite knowledge is basic probability and linear algebra

2.5.2 Definitions

The following concepts are fundamental

Stochastic Matrices

A stochastic matrix (or Markov matrix) is an $n \times n$ square matrix $P$ such that

1. each element of $P$ is nonnegative, and
2. each row of $P$ sums to one

Each row of $P$ can be regarded as a probability mass function over $n$ possible outcomes.

It is not too difficult to check\(^1\) that if $P$ is a stochastic matrix, then so is the $k$-th power $P^k$ for all $k \in \mathbb{N}$.

Markov Chains

There is a close connection between stochastic matrices and Markov chains.

To begin, let $S$ be a finite set with $n$ elements \(\{x_1, \ldots, x_n\}\).

The set $S$ is called the state space and $x_1, \ldots, x_n$ are the state values.

A Markov chain \(\{X_t\}\) on $S$ is a sequence of random variables on $S$ that have the Markov property.

This means that, for any date $t$ and any state $y \in S$,

\[
\mathbb{P}\{X_{t+1} = y \mid X_t\} = \mathbb{P}\{X_{t+1} = y \mid X_t, X_{t-1}, \ldots\}
\]

(2.48)

In other words, knowing the current state is enough to know probabilities for future states.

\(^1\) Hint: First show that if $P$ and $Q$ are stochastic matrices then so is their product to check the row sums, try postmultiplying by a column vector of ones. Finally, argue that $P^n$ is a stochastic matrix using induction.
In particular, the dynamics of a Markov chain are fully determined by the set of values

\[ P(x, y) := \mathbb{P}(X_{t+1} = y \mid X_t = x) \quad (x, y \in S) \] (2.49)

By construction,

- \( P(x, y) \) is the probability of going from \( x \) to \( y \) in one unit of time (one step)
- \( P(x, \cdot) \) is the conditional distribution of \( X_{t+1} \) given \( X_t = x \)

We can view \( P \) as a stochastic matrix where

\[ P_{ij} = P(x_i, x_j) \quad 1 \leq i, j \leq n \]

Going the other way, if we take a stochastic matrix \( P \), we can generate a Markov chain \( \{X_t\} \) as follows:

- draw \( X_0 \) from some specified distribution
- for each \( t = 0, 1, \ldots \), draw \( X_{t+1} \) from \( P(X_t, \cdot) \)

By construction, the resulting process satisfies (2.49)

**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 unemployed worker finds a job with probability \( \alpha \in (0, 1) \)
   2. An employed worker loses her job and becomes unemployed with probability \( \beta \in (0, 1) \)

In terms of a Markov model, we have

- \( S = \{1, 2\} \)
- \( P(1, 2) = \alpha \) and \( P(2, 1) = \beta \)

We can write out the transition probabilities in matrix form as

\[ P = \begin{pmatrix}
1 - \alpha & \alpha \\
\beta & 1 - \beta
\end{pmatrix} \]

Once we have the values \( \alpha \) and \( \beta \), we can address a range of questions, such as

- What is the average duration of unemployment?
- Over the long-run, what fraction of time does a worker find herself unemployed?
- Conditional on employment, what is the probability of becoming unemployed at least once over the next 12 months?

Well cover such applications below

2.5. Finite Markov Chains
Example 2

Using US unemployment data, Hamilton \cite{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 \( \text{ng} \) is normal growth, \( \text{mr} \) is mild recession, etc.

2.5.3 Simulation

One natural way to answer questions about Markov chains is to simulate them.

(To approximate the probability of event \( E \), we can simulate many times and count the fraction of times that \( E \) occurs)

Nice functionality for simulating Markov chains exists in QuantEcon.jl

- Efficient, bundled with lots of other useful routines for handling Markov chains

However, its also a good exercise to roll our own routines. Let's do that first and then come back to the methods in QuantEcon.jl

In these exercises well take the state space to be \( S = 1, \ldots, n \).
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 distribution.

For this task we’ll use `DiscreteRV` from `QuantEcon`.

```julia
using QuantEcon

ψ = [0.1, 0.9];  # Probabilities over sample space {1, 2}
d = DiscreteRV(ψ);
rand(d, 5)  # Generate 5 independent draws from ψ
```

We’ll write our code as a function that takes the following three arguments:

- A stochastic matrix $P$
- An initial state $\text{init}$
- A positive integer $\text{sample}_\text{size}$ representing the length of the time series the function should return

```julia
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] = rand(P_dist[X[t]])
    end
    return X
end
```

Let’s see how it works using the small matrix.

2.5. Finite Markov Chains
\[ P := \begin{pmatrix} 0.4 & 0.6 \\ 0.2 & 0.8 \end{pmatrix} \]  

(2.50)

As well 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]
X = mc_sample_path(P, sample_size=100000);
println(mean(X .== 1))
```

0.25171

**Using QuantEcons Routines**

As discussed above, QuantEcon.jl has routines for handling Markov chains, including simulation

Here is an illustration using the same \( P \) as the preceding example

```julia
P = [0.4 0.6; 0.2 0.8];
mc = MarkovChain(P)
X = simulate(mc, 100000);
mean(X .== 1)  \# Should be close to 0.25
```

0.25031

**Adding state values and initial conditions**

If we wish to, we can provide a specification of state values to MarkovChain

These state values can be integers, floats, or even strings

The following code illustrates

```julia
mc = MarkovChain(P, ["unemployed", "employed"])
simulate(mc, 4, init=1)  \# Start at state 1
```

4-element Array{String,1}:
"employed"
"employed"
"unemployed"
"unemployed"

```julia
simulate(mc, 4, init=2)  \# Start at state 2
```
simulate(mc, 4)  # Start with randomly chosen initial condition

simulate_indices(mc, 4)

2.5.4 Marginal Distributions

Suppose that

1. \( \{X_t\} \) is a Markov chain with stochastic matrix \( P \)
2. the distribution of \( X_t \) is known to be \( \psi_t \)

What then is the distribution of \( X_{t+1} \), or, more generally, of \( X_{t+m} \)?

Solution

Let \( \psi_t \) be the distribution of \( X_t \) for \( t = 0, 1, 2, \ldots \)

Our first aim is to find \( \psi_{t+1} \) given \( \psi_t \) and \( P \)

To begin, pick any \( y \in S \)

Using the law of total probability, we can decompose the probability that \( X_{t+1} = y \) as follows:

\[
P\{X_{t+1} = y\} = \sum_{x \in S} P\{X_{t+1} = y \mid X_t = x\} \cdot P\{X_t = x\}
\]

In words, to get the probability of being at \( y \) tomorrow, we account for all ways this can happen and sum their probabilities

Rewriting this statement in terms of marginal and conditional probabilities gives
\[ \psi_{t+1}(y) = \sum_{x \in S} P(x, y) \psi_t(x) \]

There are \( n \) such equations, one for each \( y \in S \)

If we think of \( \psi_{t+1} \) and \( \psi_t \) as row vectors (as is traditional in this literature), these \( n \) equations are summarized by the matrix expression

\[ \psi_{t+1} = \psi_t P \]  

(2.51)

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 (2.51), 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

\[ X_0 \sim \psi_0 \implies X_m \sim \psi_0 P^m \]

(2.52)

and, more generally,

\[ X_t \sim \psi_t \implies X_{t+m} \sim \psi_t P^m \]

(2.53)

**Multiple Step Transition Probabilities**

We know that the probability of transitioning from \( x \) to \( y \) in one step is \( P(x, y) \).

It turns out that the probability of transitioning from \( x \) to \( y \) in \( m \) steps is \( P^m(x, y) \), the \((x, y)\)-th element of the \( m \)-th power of \( P \).

To see why, consider again (2.53), but now with \( \psi_t \) putting all probability on state \( x \):

- \( 1 \) in the \( x \)-th position and zero elsewhere.

Inserting this into (2.53), we see that, conditional on \( X_t = x \), the distribution of \( X_{t+m} \) is the \( x \)-th row of \( P^m \).

In particular

\[ \mathbb{P}\{X_{t+m} = y\} = P^m(x, y) = (x, y)\text{-th element of } P^m \]
Example: Probability of Recession

Recall the stochastic matrix $P$ for recession and growth considered above

Suppose that the current state is unknown perhaps statistics are available only at the end of the current month

We estimate the probability that the economy is in state $x$ to be $\psi(x)$

The probability of being in recession (either mild or severe) in 6 months time is given by the inner product

$$\psi P^6 \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

Example 2: Cross-Sectional Distributions

The marginal distributions we have been studying can be viewed either as probabilities or as cross-sectional frequencies in large samples

To illustrate, recall our model of employment / unemployment dynamics for a given worker discussed above

Consider a large (i.e., tending to infinite) population of workers, each of whose lifetime experiences are described by the specified dynamics, independently of one another

Let $\psi$ be the current cross-sectional distribution over $\{1, 2\}$

- 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 workers career spent being employed and unemployed, respectively

2.5.5 Irreducibility and Aperiodicity

Irreducibility and aperiodicity are central concepts of modern Markov chain theory

Lets see what theyre about

Irreducibility

Let $P$ be a fixed stochastic matrix

Two states $x$ and $y$ are said to communicate with each other if there exist positive integers $j$ and $k$ such that

$$P^j(x, y) > 0 \quad \text{and} \quad P^k(y, x) > 0$$

In view of our discussion above, this means precisely that

- state $x$ can be reached eventually from state $y$, and
• state $y$ can be reached eventually from state $x$

The stochastic matrix $P$ is called **irreducible** if all states communicate; that is, if $x$ and $y$ communicate for all $(x, y)$ in $S \times S$

For example, consider the following transition probabilities for wealth of a fictitious set of households

![Graph showing transition probabilities](image)

We can translate this into a stochastic matrix, putting zeros where there is 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.jl's MarkovChain class

```julia
P = [0.9 0.1 0.0; 0.4 0.4 0.2; 0.1 0.1 0.8];
mc = MarkovChain(P)
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
P = [1.0 0.0 0.0; 0.1 0.8 0.1; 0.0 0.2 0.8];
mc = MarkovChain(P);
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

Well 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:

\[
P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix};
\]
\[
mc = \text{MarkovChain}(P);
\]
\[
\text{period}(mc)
\]

3

More formally, the **period** of a state \(x\) is the greatest common divisor of the set of integers

\[
D(x) := \{j \geq 1 : P^j(x, x) > 0\}
\]

In the last example, \(D(x) = \{3, 6, 9, \ldots\}\) for every state \(x\), so the period is 3

A stochastic matrix is called **aperiodic** if the period of every state is 1, and **periodic** otherwise

For example, the stochastic matrix associated with the transition probabilities below is periodic because, for example, state \(a\) has period 2

\[
P = \begin{bmatrix} 1 & 0.5 & 0.5 & 0.5 \\ 1.0 & 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 & 1.0 \\ 0.5 & 0.5 & 0.5 & 1.0 \end{bmatrix};
\]
\[
mc = \text{MarkovChain}(P);
\]
\[
\text{period}(mc)
\]

2

is_aperiodic(mc)

false
2.5.6 Stationary Distributions

As seen in (2.51), we can shift probabilities forward one unit of time via postmultiplication by $P$.

Some distributions are invariant under this updating process, for example,

\[ P = \begin{bmatrix} .4 & .6 \\ .2 & .8 \end{bmatrix}; \]
\[ \psi = [0.25, 0.75]; \]
\[ \psi' = P \psi \]

\[
\begin{bmatrix}
0.25 & 0.75
\end{bmatrix}
\]

Such distributions are called stationary, or invariant.

Formally, a distribution $\psi^\ast$ on $S$ is called stationary for $P$ if $\psi^\ast = \psi^\ast P$.

From this equality we immediately get $\psi^\ast = \psi^\ast P^t$ 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 will discuss this more in just a moment.

Mathematically, a stationary distribution is a fixed point of $P$ when $P$ is thought of as the map $\psi \mapsto \psi P$ from (row) vectors to (row) vectors.

**Theorem.** Every stochastic matrix $P$ has at least one stationary distribution.

(We are assuming here that the state space $S$ is finite; if not more assumptions are required.)

For a proof of this result you can apply Brouwer’s fixed point theorem, or see EDTC, theorem 4.3.5.

There may in fact be many stationary distributions corresponding to a given stochastic matrix $P$.

- For example, if $P$ is the identity matrix, then all distributions are stationary.

Since stationary distributions are long run equilibria, to get uniqueness we require that initial conditions are not infinitely persistent.

Infinite persistence of initial conditions occurs if certain regions of the state space cannot be accessed from other regions, which is the opposite of irreducibility.

This gives some intuition for the following fundamental theorem.

**Theorem.** If $P$ is both aperiodic and irreducible, then

1. $P$ has exactly one stationary distribution $\psi^\ast$.
2. For any initial distribution $\psi_0$, we have $\|\psi_0 P^t - \psi^\ast\| \to 0$ as $t \to \infty$.

For a proof, see, for example, theorem 5.2 of [Haggstrom02].

(Note that part 1 of the theorem requires only irreducibility, whereas part 2 requires both irreducibility and aperiodicity.)

A stochastic matrix satisfying the conditions of the theorem is sometimes called uniformly ergodic.

2.5. Finite Markov Chains

241
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 $\beta \to 0$, and to one as $\alpha \to 0$

**Calculating Stationary Distributions**

As discussed above, a given Markov matrix $P$ can have many stationary distributions. That is, there can be many row vectors $\psi$ such that $\psi = \psi P$.

In fact if $P$ has two distinct stationary distributions $\psi_1, \psi_2$ then it has infinitely many, since in this case, as you can verify,

$$\psi_3 := \lambda \psi_1 + (1 - \lambda) \psi_2$$

is a stationary distribution for $P$ for any $\lambda \in [0, 1]$. If we restrict attention to the case where only one stationary distribution exists, one option for finding it is to try to solve the linear system $\psi(I_n - P) = 0$ for $\psi$, where $I_n$ is the $n \times n$ identity. But the zero vector solves this equation. Hence we need to impose the restriction that the solution must be a probability distribution.

A suitable algorithm is implemented in QuantEcon.jl the next code block illustrates:

```
P = [.4 .6; .2 .8];
mc = MarkovChain(P);
stationary_distributions(mc)
```

1-element Array{Array{Float64,1},1}:
[0.25,0.75]

The stationary distribution is unique.
Convergence to Stationarity

Part 2 of the Markov chain convergence theorem stated above tells us that the distribution of $X_t$ converges to the stationary distribution regardless of where we start off.

This adds considerable weight to our interpretation of $\psi^*$ as a stochastic steady state.

The convergence in the theorem is illustrated in the next figure.

```julia
using Plots
pyplot()

P = [0.971 0.029 0.000
     0.145 0.778 0.077
     0.000 0.508 0.492]

ψ = [0.0 0.2 0.8]

t = 20
x vals = Array{Float64}(t+1)
y_vals = Array{Float64}(t+1)
z_vals = Array{Float64}(t+1)
colors = []

for i=1:t
    x_vals[i] = ψ[1]
    y_vals[i] = ψ[2]
    z_vals[i] = ψ[3]
    ψ = ψ*P
    push!(colors, :red)
end
push!(colors, :black)
mc = MarkovChain(P)
ψ_star = stationary_distributions(mc)[1]
x_vals[t+1] = ψ_star[1]
y_vals[t+1] = ψ_star[2]
z_vals[t+1] = ψ_star[3]
scatter(x_vals, y_vals, z_vals, color=colors)
plot!(lims=(0, 1), ticks=[0.25 0.5 0.75]', legend=:none, camera=(300, 30))
```
Here

- $P$ is the stochastic matrix for recession and growth *considered above*
- The highest red dot is an arbitrarily chosen initial probability distribution $\psi$, represented as a vector in $\mathbb{R}^3$
- The other red dots are the distributions $\psi P^t$ for $t = 1, 2, \ldots$
- The black dot is $\psi^*$

The code for the figure can be found here you might like to try experimenting with different initial conditions

### 2.5.7 Ergodicity

Under irreducibility, yet another important result obtains: For all $x \in S$,

\[
\frac{1}{n} \sum_{t=1}^{m} 1\{X_t = x\} \to \psi^*(x) \quad \text{as } m \to \infty \tag{2.54}
\]

Here

- $1\{X_t = x\} = 1$ if $X_t = x$ and zero otherwise
- convergence is with probability one
- the result does not depend on the distribution (or value) of $X_0$

The result tells us that the fraction of time the chain spends at state $x$ converges to $\psi^*(x)$ as time goes to infinity.
This gives us another way to interpret the stationary distribution provided that the convergence result in (2.54) is valid.

The convergence in (2.54) 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.

### 2.5.8 Computing Expectations

We are interested in computing expectations of the form

$$ \mathbb{E}[h(X_t)] $$

and conditional expectations such as

$$ \mathbb{E}[h(X_{t+k}) \mid X_t = x] $$

where

- $\{X_t\}$ is a Markov chain generated by $n \times n$ stochastic matrix $P$
- $h$ is a given function, which, in expressions involving matrix algebra, well think of as the column vector

$$ h = \begin{pmatrix} h(x_1) \\
\vdots \\
h(x_n) \end{pmatrix} $$
The unconditional expectation (2.55) is easy: We just sum over the distribution of $X_t$ to get

$$
\mathbb{E}[h(X_t)] = \sum_{x \in S} (\psi P^t)(x)h(x)
$$

Here $\psi$ is the distribution of $X_0$

Since $\psi$ and hence $\psi P^t$ are row vectors, we can also write this as

$$
\mathbb{E}[h(X_t)] = \psi P^t h
$$

For the conditional expectation (2.56), we need to sum over the conditional distribution of $X_{t+k}$ given $X_t = x$

We already know that this is $P^k(x, \cdot)$, so

$$
\mathbb{E}[h(X_{t+k}) \mid X_t = x] = (P^k h)(x) \quad (2.57)
$$

The vector $P^k h$ stores the conditional expectation $\mathbb{E}[h(X_{t+k}) \mid X_t = x]$ over all $x$

**Expectations of Geometric Sums**

Sometimes we also want to compute expectations of a geometric sum, such as $\sum_0^\infty \beta^j h(X_t)$

In view of the preceding discussion, this is

$$
\mathbb{E} \left[ \sum_{j=0}^\infty \beta^j h(X_{t+j}) \mid X_t = x \right] = [(I - \beta P)^{-1} h](x)
$$

where

$$(I - \beta P)^{-1} = I + \beta P + \beta^2 P^2 + \ldots$$

Premultiplication by $(I - \beta P)^{-1}$ amounts to applying the **resolvent operator**

**2.5.9 Exercises**

**Exercise 1**

According to the discussion *above*, if a workers employment dynamics obey the stochastic matrix

$$
P = \begin{pmatrix}
1 - \alpha & \alpha \\
\beta & 1 - \beta
\end{pmatrix}
$$

with $\alpha \in (0,1)$ and $\beta \in (0,1)$, then, in the long-run, the fraction of time spent unemployed will be

$$
p := \frac{\beta}{\alpha + \beta}
$$
In other words, if \( \{X_t\} \) represents the Markov chain for employment, then \( \bar{X}_m \to p \) as \( m \to \infty \), where

\[
\bar{X}_m := \frac{1}{m} \sum_{t=1}^{m} \mathbb{1}\{X_t = 1\}
\]

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 \( \bar{X}_m - p \) against \( m \), 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.

2.5. Finite Markov Chains
(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

\[
 P(i, j) = \frac{1}{\ell_i} \text{ if } i \text{ has a link to } j \\
0 \text{ otherwise}
\]

The matrix \( P \) is a stochastic matrix provided that each page has at least one link.

With this definition of \( P \) we have

\[
 r_j = \sum_{i \in L_j} \frac{r_i}{\ell_i} = \sum_{i} 1\{i \to j\} \frac{r_i}{\ell_i} = \sum_{i} P(i, j) r_i 
\]

Writing \( r \) for the row vector of rankings, this becomes \( r = rP \).

Hence \( r \) is the stationary distribution of the stochastic matrix \( P \).

Let’s think of \( P(i, j) \) as the probability of moving from page \( i \) to page \( j \).

The value \( P(i, j) \) has the interpretation

- \( P(i, j) = 1/k \) if \( i \) has \( k \) outbound links, and \( j \) is one of them
- \( P(i, j) = 0 \) if \( i \) has no direct link to \( j \)

Thus, motion from page to page is that of a web surfer who moves from one page to another by randomly clicking on one of the links on that page.

Here random means that each link is selected with equal probability.

Since \( r \) is the stationary distribution of \( P \), assuming that the uniform ergodicity condition is valid, we can interpret \( r_j \) as the fraction of time that a (very persistent) random surfer spends at page \( j \).

Your exercise is to apply this ranking algorithm to the graph pictured above, and return the list of pages ordered by rank.

The data for this graph is in the web_graph_data.txt file, you can also view it here.

There is a total of 14 nodes (i.e., web pages), the first named \( a \) and the last named \( n \).

A typical line from the file has the form...
d -> h;

This should be interpreted as meaning that there exists a link from d to h.

To parse this file and extract the relevant information, you can use regular expressions.

The following code snippet provides a hint as to how you can go about this:

```plaintext
matchall(r"\w", "x +++ y ***** z")

3-element Array{SubString{String},1}:
  "x"
  "y"
  "z"

matchall(r"\w", "a ^ ^ b &&& \$\$ c")

3-element Array{SubString{String},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

\[ \sigma_y^2 := \frac{\sigma_u^2}{1 - \rho^2} \]

Tauchens method \cite{Tau86} is the most common method for approximating this continuous state process with a finite state Markov chain.

A routine for this already exists in `QuantEcon.jl` but lets write our own version as an exercise.

As a first step we choose

- \( n \), the number of states for the discrete approximation
- \( m \), an integer that parameterizes the width of the state space

Next we create a state space \( \{ x_0, \ldots, x_{n-1} \} \subset \mathbb{R} \) and a stochastic \( n \times n \) matrix \( P \) such that
• $x_0 = -m \sigma_y$
• $x_{n-1} = m \sigma_y$
• $x_{i+1} = x_i + s$ where $s = (x_{n-1} - x_0)/(n - 1)$

Let $F$ be the cumulative distribution function of the normal distribution $N(0, \sigma_u^2)$

The values $P(x_i, x_j)$ are computed to approximate the AR(1) process omitting the derivation, the rules are as follows:

1. If $j = 0$, then set
   \[ P(x_i, x_j) = P(x_i, x_0) = F(x_0 - \rho x_i + s/2) \]

2. If $j = n - 1$, then set
   \[ P(x_i, x_j) = P(x_i, x_{n-1}) = 1 - F(x_{n-1} - \rho x_i - s/2) \]

3. Otherwise, set
   \[ P(x_i, x_j) = F(x_j - \rho x_i + s/2) - F(x_j - \rho x_i - s/2) \]

The exercise is to write a function `approx_markov(rho, sigma_u, m=3, n=7)` that returns $\{x_0, \ldots, x_{n-1}\} \subset \mathbb{R}$ and $n \times n$ matrix $P$ as described above

• Even better, write a function that returns an instance of `QuantEcon.jls MarkovChain` type

### 2.5.10 Solutions

```plaintext
careful: P and p are distinct
```

**Exercise 1**

Compute the fraction of time that the worker spends unemployed, and compare it to the stationary probability.

```plaintext
\begin{align*}
\alpha & = \beta = 0.1 \\
N & = 10000 \\
p & = \beta / (\alpha + \beta) \\
P & = [1 - \alpha \quad \alpha] \quad \beta \quad 1 - \beta]
\end{align*}
```

```plaintext
mc = MarkovChain(P)
```

Discrete Markov Chain
stochastic matrix of type Array{Float64,2}:
[0.9 0.1; 0.1 0.9]

### 2.5. Finite Markov Chains
labels = []
y_vals = []

for x0 = 1:2
    # == Generate time series for worker that starts at x0 ==#
    X = simulate_indices(mc, N; init=x0)

    # == Compute fraction of time spent unemployed, for each n ==#
    X_bar = cumsum(X .== 1) ./ (collect(1:N))

    l = LaTeXString("\$X_0 = \$x0\$")
    push!(labels, l)
    push!(y_vals, X_bar - p)
end

plot(y_vals, color=[:blue :green], fillrange=0, fillalpha=0.1,
    ylims=(-0.25, 0.25), label=reshape(labels,1,length(labels)))

**Exercise 2**

First save the data into a file called `web_graph_data.txt` by executing the next cell

```julia
f = open("web_graph_data.txt", "w")
contents = """a -> d;
a -> f;
b -> j;"
f.write(contents)
f.close()
```
QuantEcon.lectures-julia PDF, Release 2018-Aug-8

2.5. Finite Markov Chains

```
# b -> k;
b -> m;
c -> c;
c -> g;
c -> j;
c -> m;
d -> f;
d -> h;
d -> k;
e -> d;
e -> h;
e -> l;
f -> a;
f -> b;
f -> j;
f -> l;
g -> b;
g -> j;
h -> d;
h -> g;
h -> l;
h -> m;
i -> g;
i -> h;
i -> n;
j -> e;
j -> i;
j -> k;
k -> n;
l -> m;
m -> g;
n -> c;
n -> j;
n -> m;
"
write(f, contents)
close(f)
```

```
#= 
Return list of pages, ordered by rank
#

infile = "web_graph_data.txt"
alphabet = "abcdefghijklmnopqrstuvwxyz"

n = 14 # Total number of web pages (nodes)

# == Create a matrix Q indicating existence of links == #
# * Q[i, j] = 1 if there is a link from i to j
# * Q[i, j] = 0 otherwise
Q = zeros(Int64, n, n)
f = open(infile, "r")
edges = readlines(f)
```
close(f)
for edge in edges
    from_node, to_node = matchall(r"\w", edge)
    i = searchindex(alphabet, from_node)
    j = searchindex(alphabet, to_node)
    Q[i, j] = 1
end

# == Create the corresponding Markov matrix P == #
P = Array{Float64}(n, n)
for i=1:n
    P[i, :] = Q[i, :] / sum(Q[i, :])
end

mc = MarkovChain(P)

# == Compute the stationary distribution r == #
r = stationary_distributions(mc)[1]
ranked_pages = Dict(alphabet[i] => r[i] for i=1:n)

# == Print solution, sorted from highest to lowest rank == #
println("Rankings
***")
sort_inds = reverse!(sortperm(collect(values(ranked_pages))))
the_keys = collect(keys(ranked_pages))
the_vals = collect(values(ranked_pages))
for i in sort_inds
    @printf("%s: %.4f\n", the_keys[i], the_vals[i])
end

Rankings
***
g: 0.1607
j: 0.1594
m: 0.1195
n: 0.1088
k: 0.0911
b: 0.0833
i: 0.0531
e: 0.0531
c: 0.0483
h: 0.0456
l: 0.0320
d: 0.0306
f: 0.0116
a: 0.0029

Exercise 3

A solution from QuantEcon.jl can be found here
2.6 Continuous State Markov Chains

Contents

- Continuous State Markov Chains
  - Overview
  - The Density Case
  - Beyond Densities
  - Stability
  - Exercises
  - Solutions
  - Appendix

2.6.1 Overview

In a previous lecture we learned about finite Markov chains, a relatively elementary class of stochastic dynamic models.

The present lecture extends this analysis to continuous (i.e., uncountable) state Markov chains.

Most stochastic dynamic models studied by economists either fit directly into this class or can be represented as continuous state Markov chains after minor modifications.

In this lecture, our focus will be on continuous Markov models that

- evolve in discrete time
- are often nonlinear

The fact that we accommodate nonlinear models here is significant, because linear stochastic models have their own highly developed tool set, as well see later on.

The question that interests us most is: Given a particular stochastic dynamic model, how will the state of the system evolve over time?

In particular,

- What happens to the distribution of the state variables?
- Is there anything we can say about the average behavior of these variables?
- Is there a notion of steady state or long run equilibrium that'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.

2.6.2 The Density Case

You are probably aware that some distributions can be represented by densities and some cannot. (For example, distributions on the real numbers $\mathbb{R}$ that put positive probability on individual points have no density representation.)

We are going to start our analysis by looking at Markov chains where the one step transition probabilities have density representations.

The benefit is that the density case offers a very direct parallel to the finite case in terms of notation and intuition.

Once we've built some intuition we'll cover the general case.

Definitions and Basic Properties

In our lecture on finite Markov chains, we studied discrete time Markov chains that evolve on a finite state space $S$.

In this setting, the dynamics of the model are described by a stochastic matrix $P$ that is a nonnegative square matrix $P = P[i,j]$ such that each row $P[i,:]$ 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 $P[i,:]$, one for each $i \in S$
- $P[i,:]$ is the distribution of $X_{t+1}$ given $X_t = i$

(As you probably recall, when using Julia arrays, $P[i,:]$ 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 $p: S \times S \to \mathbb{R}$ with the property that

1. $p(x, y) \geq 0$ for all $x, y \in S$
2. $\int p(x, y) dy = 1$ for all $x \in S$

(Integrals are over the whole space unless otherwise specified)

For example, let $S = \mathbb{R}$ and consider the particular stochastic kernel $p_w$ defined by

$$p_w(x, y) := \frac{1}{\sqrt{2\pi}} \exp\left\{ \frac{-(y - x)^2}{2} \right\}$$

(2.58)

What kind of model does $p_w$ represent?

The answer is, the (normally distributed) random walk

$$X_{t+1} = X_t + \xi_{t+1} \quad \text{where} \quad \{\xi_t\} \overset{iid}{\sim} N(0, 1)$$

(2.59)

To see this, let's find the stochastic kernel $p$ corresponding to (2.59)

Recall that $p(x, \cdot)$ represents the distribution of $X_{t+1}$ given $X_t = x$

Letting $X_t = x$ in (2.59) 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 (2.58)

### Connection to Stochastic Difference Equations

In the previous section, we made the connection between stochastic difference equation (2.59) and stochastic kernel (2.58)

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

$$X_{t+1} = \mu(X_t) + \sigma(X_t) \xi_{t+1}$$

(2.60)

Here we assume that

- $\{\xi_t\} \overset{iid}{\sim} \phi$, 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 (2.59) is a special case of (2.60), 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

\[
X_{t+1} = \alpha X_t + (\beta + \gamma X_t^2)^{1/2} \xi_{t+1}\tag{2.61}
\]

This is a special case of (2.60) with \( \mu(x) = \alpha x \) and \( \sigma(x) = (\beta + \gamma x^2)^{1/2} \)

**Example 3:** With stochastic production and a constant savings rate, the one-sector neoclassical growth model leads to a law of motion for capital per worker such as

\[
k_{t+1} = s A_{t+1} f(k_t) + (1 - \delta) k_t\tag{2.62}
\]

Here

• \( s \) is the rate of savings
• \( A_{t+1} \) is a production shock
  – The \( t + 1 \) subscript indicates that \( A_{t+1} \) is not visible at time \( t \)
• \( \delta \) is a depreciation rate
• \( f : \mathbb{R}_+ \to \mathbb{R}_+ \) is a production function satisfying \( f(k) > 0 \) whenever \( k > 0 \)

(The fixed savings rate can be rationalized as the optimal policy for a particular set of technologies and preferences (see [LS18], section 3.1.2), although we omit the details here)

Equation (2.62) is a special case of (2.60) with \( \mu(x) = (1 - \delta)x \) and \( \sigma(x) = sf(x) \)

Now let’s obtain the stochastic kernel corresponding to the generic model (2.60)

To find it, note first that if \( U \) is a random variable with density \( f_U \), and \( V = a + bU \) for some constants \( a, b \) with \( b > 0 \), then the density of \( V \) is given by

\[
f_V(v) = \frac{1}{b} f_U \left( \frac{v - a}{b} \right)\tag{2.63}
\]

(The proof is below. For a multidimensional version see EDTC, theorem 8.1.3)

Taking (2.63) as given for the moment, we can obtain the stochastic kernel \( p \) for (2.60) 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 (2.63),
$p(x, y) = \frac{1}{\sigma(x)} \phi \left( \frac{y - \mu(x)}{\sigma(x)} \right)$  \hfill (2.64)

For example, the growth model in (2.62) has stochastic kernel

$p(x, y) = \frac{1}{sf(x)} \phi \left( \frac{y - (1 - \delta)x}{sf(x)} \right)$  \hfill (2.65)

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

$$
\psi_{t+1}(y) = \int p(x, y) \psi_t(x) \, dx, \quad \forall y \in S
$$  \hfill (2.66)

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

$$
(\psi P)(y) = \int p(x, y) \psi(x) \, dx
$$  \hfill (2.67)

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 (2.66) more succinctly as $\psi_{t+1}(y) = (\psi_t P)(y)$ for all $y$, or, dropping the $y$ and letting $=$ indicate equality of functions,

$$\psi_{t+1} = \psi_t P$$

Equation (2.68) 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 (2.68) is a deterministic difference equation.

Thus, by converting a stochastic difference equation such as (2.60) 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 (2.67) and (2.68) using numerical integration.

However, to produce $\psi P$ from $\psi$ via (2.67), 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:

$$k_{t+1} = sA_{t+1}f(k_t) + (1 - \delta)k_t$$

(2.69)

Our aim is to compute the sequence $\{\psi_t\}$ associated with this model and fixed initial condition $\psi_0$.

To approximate $\psi_t$ by simulation, recall that, by definition, $\psi_t$ is the density of $k_t$ given $k_0 \sim \psi_0$. 
If we wish to generate observations of this random variable, all we need to do is

1. draw $k_0$ from the specified initial condition $\psi_0$
2. draw the shocks $A_1, \ldots, A_t$ from their specified density $\phi$
3. compute $k_t$ iteratively via (2.69)

If we repeat this $n$ times, we get $n$ independent observations $k^1_t, \ldots, k^n_t$

With these draws in hand, the next step is to generate some kind of representation of their distribution $\psi_t$.

A naive approach would be to use a histogram, or perhaps a smoothed histogram using 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 $k^1_{t-1}, \ldots, k^n_{t-1}$ and form the estimate

$$
\psi^n_t(y) = \frac{1}{n} \sum_{i=1}^{n} p(k^i_{t-1}, y)
$$

where $p$ is the growth model stochastic kernel in (2.65).

What is the justification for this slightly surprising estimator?

The idea is that, by the strong law of large numbers,

$$
\frac{1}{n} \sum_{i=1}^{n} p(k^i_{t-1}, y) \to \mathbb{E} p(k^i_{t-1}, y) = \int p(x, y) \psi_{t-1}(x) \, dx = \psi_t(y)
$$

with probability one as $n \to \infty$.

Here the first equality is by the definition of $\psi_{t-1}$, and the second is by (2.66).

We have just shown that our estimator $\psi^n_t(y)$ in (2.70) 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 function which calls an LAE type for estimating densities by this technique can be found in `lae.jl`.

This function returns the right-hand side of (2.70) 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

The following code is an example of usage for the stochastic growth model described above.

```julia
#=
@author: Spencer Lyon <spencer.lyon@nyu.edu>
    Victoria Gregory <victoria.gregory@nyu.edu>

#= using QuantEcon
    using Distributions
    using Plots
    using LaTeXStrings

s = 0.2
δ = 0.1
a_σ = 0.4
    # A = exp(B) where B ~ N(0, a_σ)
α = 0.4
    # We set f(k) = k**α
ψ_0 = Beta(5.0, 5.0)
    # Initial distribution
= LogNormal(0.0, a_σ)

function p(x, y)
    #=
        Stochastic kernel for the growth model with Cobb-Douglas production.
        Both x and y must be strictly positive.
    =#
        d = s * x.^α

        # scipy silently evaluates the pdf of the lognormal dist at a negative
        # value as zero. It should be undefined and Julia recognizes this.
        pdf_arg = clamp.((y - (1-δ) .* x) ./ d, eps(), Inf)
        return pdf.(ψ_0, pdf_arg) ./ d
end

n = 10000  # Number of observations at each date t
T = 30     # Compute density of k_t at 1,...,T+1

# Generate matrix s.t. t-th column is n observations of k_t
k = Array{Float64}(n, T)
A = rand!(k, Array{Float64}(n, T))

# Draw first column from initial distribution
k[:, 1] = rand(ψ_0, n) ./ 2  # divide by 2 to match scale=0.5 in py version
    for t=1:T-1
        k[:, t+1] = s*A[:, t] .* k[:, t].^α + (1-δ) .* k[:, t]
    end

# Generate T instances of LAE using this data, one for each date t
laes = [LAE(p, k[:, t]) for t=T:-1:1]
```
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).
2.6.3 Beyond Densities

Up until now, we have focused exclusively on continuous state Markov chains where all conditional distributions $p(x, \cdot)$ are densities.

As discussed above, not all distributions can be represented as densities.

If the conditional distribution of $X_{t+1}$ given $X_t = x$ cannot be represented as a density for some $x \in S$, then we need a slightly different theory.

The ultimate option is to switch from densities to probability measures, but not all readers will be familiar with measure theory.

We can, however, construct a fairly general theory using distribution functions.

Example and Definitions

To illustrate the issues, recall that Hopenhayn and Rogerson [HR93] study a model of firm dynamics where individual firm productivity follows the exogenous process

$$X_{t+1} = a + \rho X_t + \xi_{t+1}, \quad \text{where} \quad \{\xi_t\} \overset{\text{IID}}{\sim} N(0, \sigma^2)$$

As is, this fits into the density case we treated above.

However, the authors wanted this process to take values in $[0, 1]$, so they added boundaries at the end points $0$ and $1$.

One way to write this is

$$X_{t+1} = h(a + \rho X_t + \xi_{t+1}) \quad \text{where} \quad h(x) := x 1\{0 \leq x \leq 1\} + 1\{x > 1\}$$

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 (2.66) are then replaced by

$$F_{t+1}(y) = \int G(x, y) F_t(dx) \quad (2.71)$$

Here $F_t$ and $F_{t+1}$ are cdfs representing the distribution of the current state and next period state.

The intuition behind (2.71) is essentially the same as for (2.66).
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.

2.6.4 Stability

In our lecture on finite Markov chains we also studied stationarity, stability and ergodicity. Here we will cover the same topics for the continuous case.

We will, however, treat only the density case (as in this section), where the stochastic kernel is a family of densities. The general case is relatively similar; references are given below.

Theoretical Results

Analogous to the finite case, given a stochastic kernel $p$ and corresponding Markov operator as defined in (2.67), a density $\psi^*$ on $S$ is called stationary for $P$ if it is a fixed point of the operator $P$.

In other words,

$$\psi^*(y) = \int p(x, y)\psi^*(x) \, dx, \quad \forall y \in S$$  \hspace{1cm} (2.72)

As with the finite case, if $\psi^*$ is stationary for $P$, and the distribution of $X_0$ is $\psi^*$, then, in view of (2.68), $X_t$ 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 ($\psi \in D$ and $\psi P = \psi \Rightarrow \psi = \psi^*$), and also global convergence in the sense that

$$\forall \psi \in D, \quad \psi P^t \to \psi^* \quad \text{as} \quad t \to \infty$$  \hspace{1cm} (2.73)

This combination of existence, uniqueness and global convergence in the sense of (2.73) is often referred to as global stability.
Under very similar conditions, we get *ergodicity*, which means that

\[
\frac{1}{n} \sum_{t=1}^{n} h(X_t) \to \int h(x) \psi^*(x) dx \quad \text{as } n \to \infty
\]

(2.74)

for any (measurable) function \( h: S \to \mathbb{R} \) such that the right-hand side is finite.

Note that the convergence in (2.74) 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 (2.74) 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 (2.73) 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 (2.60) that is stable and ergodic.

Let \( p \) be the corresponding stochastic kernel, as given in (2.64).

To approximate the stationary density \( \psi^* \), we can simply generate a long time series \( X_0, X_1, \ldots, X_n \) and estimate \( \psi^* \) via...
\[ \psi_n^*(y) = \frac{1}{n} \sum_{t=1}^{n} p(X_t, y) \]  
(2.75)

This is essentially the same as the look ahead estimator (2.70), except that now the observations we generate are a single time series, rather than a cross section.

The justification for (2.75) is that, with probability one as \( n \to \infty \),

\[ \frac{1}{n} \sum_{t=1}^{n} p(X_t, y) \to \int p(x, y) \psi^*(x) \, dx = \psi^*(y) \]

where the convergence is by (2.74) and the equality on the right is by (2.72).

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.

### 2.6.5 Exercises

**Exercise 1**

Consider the simple threshold autoregressive model

\[ X_{t+1} = \theta |X_t| + (1 - \theta^2)^{1/2} \xi_{t+1} \]

where \( \{\xi_t\} \text{ i.i.d. } N(0, 1) \)  
(2.76)

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

\[ \psi^*(y) = 2 \phi(y) \Phi \left[ \frac{\theta y}{(1 - \theta^2)^{1/2}} \right] \]

(2.77)

Here \( \phi \) is the standard normal density and \( \Phi \) is the standard normal c.d.f.

As an exercise, compute the look ahead estimate of \( \psi^* \), as defined in (2.75), and compare it with \( \psi^* \) in (2.77) to see whether they are indeed close for large \( n \).

In doing so, set \( \theta = 0.8 \) and \( n = 500 \).

The next figure shows the result of such a computation.
The additional density (black line) is a nonparametric kernel density estimate, added to the solution for illustration.

(You can try to replicate it before looking at the solution if you want to)

As you can see, the look ahead estimator is a much tighter fit than the kernel density estimator.

If you repeat the simulation you will see that this is consistently the case.

**Exercise 2**

Replicate the figure on global convergence *shown above*.

The densities come from the stochastic growth model treated *at the start of the lecture*.

Begin with the code found in *stochasticgrowth.py*.

Use the same parameters.

For the four initial distributions, use the beta distribution and shift the random draws as shown below:

```
ψ_0 = Beta(5.0, 5.0)  # Initial distribution
n = 1000
# .... more setup
for i=1:4
```
Exercise 3

A common way to compare distributions visually is with boxplots. To illustrate, let’s generate three artificial data sets and compare them with a boxplot.

```julia
using StatPlots  # needed for box plot support

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)
data = vcat(x, y, z)
l = [LaTeXString("\$X\$") LaTeXString("\$Y\$") LaTeXString("\$Z\$") ]
xlabels = reshape(repmat(l, n), n*3, 1)

boxplot(xlabels, data, label="", ylims=(-2, 14))
```

The three data sets are

\[
\{X_1, \ldots, X_n\} \sim LN(0, 1), \ \{Y_1, \ldots, Y_n\} \sim N(2, 1), \text{ and } \{Z_1, \ldots, Z_n\} \sim N(4, 1),
\]

The figure looks as follows:
Each data set is represented by a box, where the top and bottom of the box are the third and first quartiles of the data, and the red line in the center is the median.

The boxes give some indication as to

- the location of probability mass for each sample
- whether the distribution is right-skewed (as is the lognormal distribution), etc

Now let's put these ideas to use in a simulation.

Consider the threshold autoregressive model in (2.76)

We know that the distribution of $X_t$ will converge to (2.77) whenever $|\theta| < 1$

Let's observe this convergence from different initial conditions using boxplots.

In particular, the exercise is to generate $J$ boxplot figures, one for each initial condition $X_0$ in

```python
initial_conditions = 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 (2.76)
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

**Exercise 1**

Look ahead estimation of a TAR stationary density, where the TAR model is

\[ X_{t+1} = \theta |X_t| + (1 - \theta^2)^{1/2} \xi_{t+1} \]

and \( \xi_t \sim N(0, 1) \). Try running at \( n = 10, 100, 1000, 10000 \) to get an idea of the speed of convergence.

```julia
using KernelDensity

= Normal(0.0, 1.0)
n = 500
\( \theta = 0.8 \)
d = sqrt(1.0 - \( \theta^2 \))
\( \delta = \theta / d \)
srand(41)  # reproducible results

# true density of TAR model
_\_star(y) = 2 .* pdf.((, y) .* cdf.(, \( \delta \) * y)

# Stochastic kernel for the TAR model.
p_TAR(x, y) = pdf.(,(y - \( \theta \) .* abs.(x)) ./ d) ./ d

Z = rand(, n)
X = zeros(n)
for t=1:n-1
    X[t+1] = \( \theta \) * abs(X[t]) + d * Z[t]
end

_\_est(a) = lae_est(LAE(p_TAR, X), a)
k_est = kde(X)

ys = linspace(-3, 3, 200)
plot(ys, _\_star(ys), color=:blue, lw=2, alpha=0.6, label="true")
plot!(ys, _\_est(ys), color=:green, lw=2, alpha=0.6, label="look ahead estimate")
plot!(k_est.x, k_est.density, color=:black, lw=2, alpha=0.6, label="kernel-based estimate")
```
Exercise 2

Here's one program that does the job.

```
s = 0.2
d = 0.1
a_σ = 0.4  # A = exp(B) where B ~ N(0, a_σ)
α = 0.4    # We set f(k) = k**α
ψ_0 = Beta(5.0, 5.0)  # Initial distribution
    = LogNormal(0.0, a_σ)
srand(42)  # reproducible results

function p_growth(x, y)
    # Stochastic kernel for the growth model with Cobb-Douglas production.
    # Both x and y must be strictly positive.
    #
    d = s * x.^α

    # scipy silently evaluates the pdf of the lognormal dist at a negative
    # value as zero. It should be undefined and Julia recognizes this.
    pdf_arg = clamp.((y .- (1-δ) .* x) ./ d, eps(), Inf)
    return pdf.(ψ, pdf_arg) ./ d
end
```
n = 1000  # Number of observations at each date t
T = 40    # Compute density of k_t at 1,...,T+1

xmax = 6.5
ygrid = linspace(0.01, xmax, 150)
laes_plot = zeros(length(ygrid), 4*T)
colors = []

for i=1:4
    k = Array{Float64}(n, T)
    A = rand!(, Array{Float64}(n, T))

    # Draw first column from initial distribution
    # match scale=0.5 and loc=2*i in julia version
    k[:, 1] = (rand(_0, n) .+ 2.5i) ./ 2
    for t=1:T-1
        k[:, t+1] = s*A[:, t] .* k[:, t].^α + (1-δ) .* k[:, t]
    end

    # Generate T instances of LAE using this data, one for each date t
    laes = [LAE(p_growth, k[:, t]) for t=T:-1:1]
    ind = i
    for j = 1:T
        ψ = laes[j]
        laes_plot[:, ind] = lae_est(ψ, ygrid)
        ind = ind + 4
        push!(colors, RGBA(0, 0, 0, 1 - (j - 1)/T))
    end
end

#colors = reshape(reshape(colors, T, 4)' , 4*T, 1)

plot(ygrid, laes_plot, layout=(2,2), color=colors,
     legend=:none, xlabel="capital", xlims=(0, xmax))
Exercise 3

Here's a possible solution.

Note the way we use vectorized code to simulate the $k$ time series for one boxplot all at once.

```julia
n = 20
k = 5000
J = 6
srand(43)  # reproducible results
θ = 0.9
d = sqrt(1 - θ^2)
δ = θ / d

initial_conditions = linspace(8, 0, J)

Z = randn(k, n, J)
titles = []
data = []
x_labels = []
for j=1:J
    title = "time series from t = \$(initial_conditions[j])\"
    push!(titles, title)

    X = Array{Float64}(k, n)
    X[:, 1] = initial_conditions[j]
    labels = []
```

2.6. Continuous State Markov Chains
labels = vcat(labels, ones(k, 1))
for t=2:n
    X[:, t] = \theta \cdot \text{abs}(X[:, t-1]) + d \cdot Z[:, t, j]
    labels = vcat(labels, t*ones(k, 1))
end
X = reshape(X, n*k, 1)
push!(data, X)
push!(x_labels, labels)
end

boxplot(x_labels, data, layout=(J, 1), title=reshape(titles, 1, length(titles)), ylims=(-4, 8), legend=:none, yticks=-4:2:8, xticks=1:20)
plot!(size=(800, 2000))
2.6. Continuous State Markov Chains

time series from $t = 8.0$

time series from $t = 6.4$

time series from $t = 4.8$

time series from $t = 3.2$
2.6.7 Appendix

Here's the proof of (2.63)

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 (2.63)

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

2.7.1 Overview

This lecture provides a simple and intuitive introduction to the Kalman filter, for those who either

- have heard of the Kalman filter but dont know how it works, or
- know the Kalman filter equations, but dont know where they come from

For additional (more advanced) reading on the Kalman filter, see

- [LS18], section 2.7.
- [AM05]

The second reference presents a comprehensive treatment of the Kalman filter

Required knowledge: Familiarity with matrix manipulations, multivariate normal distributions, covariance matrices, etc.

2.7.2 The Basic Idea

The Kalman filter has many applications in economics, but for now lets pretend that we are rocket scientists
A missile has been launched from country Y and our mission is to track it

Let \( x \in \mathbb{R}^2 \) denote the current location of the missile, a pair indicating latitude-longitude coordinates on a map.

At the present moment in time, the precise location \( x \) is unknown, but we do have some beliefs about \( x \).

One way to summarize our knowledge is a point prediction \( \hat{x} \)

- But what if the President wants to know the probability that the missile is currently over the Sea of Japan?
- Then it is better to summarize our initial beliefs with a bivariate probability density \( p \)

\[
\int_E p(x) \, dx \quad \text{indicates the probability that we attach to the missile being in region } E
\]

The density \( p \) is called our prior for the random variable \( x \).

To keep things tractable in our example, we assume that our prior is Gaussian. In particular, we take

\[
p = N(\hat{x}, \Sigma) \tag{2.78}
\]

where \( \hat{x} \) is the mean of the distribution and \( \Sigma \) is a \( 2 \times 2 \) covariance matrix. In our simulations, we will suppose that

\[
\hat{x} = \begin{pmatrix} 0.2 \\ -0.2 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} 0.4 & 0.3 \\ 0.3 & 0.45 \end{pmatrix} \tag{2.79}
\]

This density \( p(x) \) is shown below as a contour map, with the center of the red ellipse being equal to \( \hat{x} \)

```julia
@file
@author : Spencer Lyon <spencer.lyon@nyu.edu>
   Victoria Gregory <victoria.gregory@nyu.edu>

using Plots
pyplot()
using LaTeXStrings

function bivariate_normal(X::Matrix,
   Y::Matrix,
   _x::Real=1.0,
   _y::Real=1.0,
   _x::Real=0.0,
   _y::Real=0.0,
   _xy::Real=0.0)

   Xμ = X .- _x
   Yμ = Y .- _y

   ρ = _xy / (_x * _y)
```

2.7. A First Look at the Kalman Filter 279
\[ z = \frac{X\mu.^2 + Y\mu.^2}{\sqrt{\varSigma}} + \frac{2}{\sqrt{\varSigma}} - 2 \ast \rho \ast X\mu \ast Y\mu / (\sigma_x \ast \sigma_y) \]

\[ \text{denom} = 2\pi \ast \sigma_x \ast \sigma_y \ast \sqrt{1 - \rho^2} \]

\[ \text{return} \quad \exp(-z / (2 \ast (1 - \rho^2))) \ast \text{denom} \]

end

# == Set up the Gaussian prior density \( p \) == #
\[ \Sigma = \begin{bmatrix} 0.4 & 0.3 \\ 0.3 & 0.45 \end{bmatrix} \]
\[ x_{\text{hat}} = \begin{bmatrix} 0.2 \\ -0.2 \end{bmatrix} \]

# == Define the matrices \( G \) and \( R \) from the equation \( y = G x + N(0, R) \) == #
\[ G = \text{eye}(2) \]
\[ R = 0.5 \ast \Sigma \]

# == The matrices \( A \) and \( Q \) == #
\[ A = \begin{bmatrix} 1.2 & 0 \\ 0 & -0.2 \end{bmatrix} \]
\[ Q = 0.3 \ast \Sigma \]

# == The observed value of \( y \) == #
\[ y = \begin{bmatrix} 2.3 \\ -1.9 \end{bmatrix} \]

# == Set up grid for plotting == #
\[ x_{\text{grid}} = \text{linspace}(-1.5, \ 2.9, \ 100) \]
\[ y_{\text{grid}} = \text{linspace}(-3.1, \ 1.7, \ 100) \]
\[ X = \text{repmat}(x_{\text{grid}}', \ \text{length}(y_{\text{grid}}), \ 1) \]
\[ Y = \text{repmat}(y_{\text{grid}}, \ 1, \ \text{length}(y_{\text{grid}})) \]

function gen_gaussian_plot_vals(\( \mu, \ C \))
    "Z values for plotting the bivariate Gaussian \( N(\mu, \ C) \)"
    \[ m_x, m_y = \mu[1], \mu[2] \]
    \[ s_x, s_y = \sqrt{\varSigma}[1], \sqrt{\varSigma}[2] \]
    \[ s_{xy} = \varSigma[1, 2] \]
    \[ \text{return bivariate_normal}(X, \ Y, \ s_x, \ s_y, \ m_x, \ m_y, \ s_{xy}) \]
end

# == Plot the figure == #
\[ Z = \text{gen_gaussian_plot_vals}(x_{\text{hat}}, \ \Sigma) \]
\[ \text{contour}(x_{\text{grid}}, \ y_{\text{grid}}, \ Z, \text{fill=\text{true}}, \text{levels=6}, \text{color=\text{:lightrainbow}}, \text{alpha=0.6}) \]
\[ \text{contour!}(x_{\text{grid}}, \ y_{\text{grid}}, \ Z, \text{fill=\text{false}}, \text{levels=6}, \text{color=\text{:grays}}, \text{cbar=\text{false}}) \]
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 \( y \).

```julia
# == Plot the figure == #
Z = gen_gaussian_plot_vals(x_hat, Σ)
contour(x_grid, y_grid, Z, fill=true, levels=6, color=:lightrainbow, alpha=0.6)
contour!(x_grid, y_grid, Z, fill=false, levels=6, color=:grays, cbar=false)
annotate!(y[1], y[2], L"$y$", color=:black)
```
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 $y = Gx + v$; where $v \sim N(0, R)$

Here $G$ and $R$ are $2 \times 2$ matrices with $R$ positive definite. Both are assumed known, and the noise term $v$ is assumed to be independent of $x$

How then should we combine our prior $p(x) = N(\hat{x}, \Sigma)$ and this new information $y$ to improve our understanding of the location of the missile?

As you may have guessed, the answer is to use Bayes theorem, which tells us to update our prior $p(x)$ to $p(x \mid y)$ via

$$p(x \mid y) = \frac{p(y \mid x) p(x)}{p(y)}$$

where $p(y) = \int p(y \mid x) p(x) dx$

In solving for $p(x \mid y)$, we observe that

- $p(x) = N(\hat{x}, \Sigma)$
- In view of (2.80), the conditional density $p(y \mid x)$ is $N(Gx, R)$
• $p(y)$ does not depend on $x$, and enters into the calculations only as a normalizing constant

Because we are in a linear and Gaussian framework, the updated density can be computed by calculating population linear regressions

In particular, the solution is known\(^1\) to be

$$p(x \mid y) = N(\hat{x}^F, \Sigma^F)$$

where

$$\hat{x}^F := \hat{x} + \Sigma G'(G\Sigma G' + R)^{-1}(y - G\hat{x}) \quad \text{and} \quad \Sigma^F := \Sigma - \Sigma G'(G\Sigma G' + R)^{-1}G\Sigma \quad (2.81)$$

Here $\Sigma G'(G\Sigma G' + R)^{-1}$ is the matrix of population regression coefficients of the hidden object $x - \hat{x}$ on the surprise $y - G\hat{x}$

This new density $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ is shown in the next figure via contour lines and the color map

The original density is left in as contour lines for comparison

\[^1\] See, for example, page 93 of [Bis06]. To get from his expressions to the ones used above, you will also need to apply the Woodbury matrix identity.
Our new density twists the prior \( p(x) \) in a direction determined by the new information \( y - G\hat{x} \n\)

In generating the figure, we set \( G \) to the identity matrix and \( R = 0.5\Sigma \) for \( \Sigma \) defined in (2.79).

**The Forecast Step**

What have we achieved so far?

We have obtained probabilities for the current location of the state (missile) given prior and current information

This is called filtering rather than forecasting, because we are filtering out noise rather than looking into the future

- \( p(x | y) = N(\hat{x}^F, \Sigma^F) \) is called the filtering distribution

But now let’s suppose that we are given another task: to predict the location of the missile after one unit of time (whatever that may be) has elapsed

To do this we need a model of how the state evolves

Let’s suppose that we have one, and that its linear and Gaussian. In particular,

\[
x_{t+1} = Ax_t + w_{t+1}, \quad \text{where} \quad w_t \sim N(0, Q)
\]  

(2.82)
Our aim is to combine this law of motion and our current distribution \( p(x | y) = N(\ddot{x}, \Sigma) \) to come up with a new predictive distribution for the location in one unit of time.

In view of (2.82), all we have to do is introduce a random vector \( x^F \sim N(\ddot{x}, \Sigma) \) 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 (2.81) tell us that

\[
\mathbb{E}[Ax^F + w] = A\mathbb{E}x^F + \mathbb{E}w = A\ddot{x} + A\Sigma G'(G\Sigma G' + R)^{-1}(y - G\ddot{x})
\]

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 \( \ddot{x} \).

Using this notation, we can summarize our results as follows.

Our updated prediction is the density \( N(\hat{x}_{\text{new}}, \Sigma_{\text{new}}) \) where

\[
\hat{x}_{\text{new}} := A\ddot{x} + K_{\Sigma}(y - G\ddot{x})
\]
\[
\Sigma_{\text{new}} := A\Sigma A' - K_{\Sigma}G\Sigma A' + Q
\]

- The density \( p_{\text{new}}(x) = N(\hat{x}_{\text{new}}, \Sigma_{\text{new}}) \) is called the predictive distribution.

The predictive distribution is the new density shown in the following figure, where the update has used parameters

\[
A = \begin{pmatrix}
1.2 & 0.0 \\
0.0 & -0.2
\end{pmatrix}, \quad Q = 0.3 * \Sigma
\]
The Recursive Procedure

Let's look back at what we've done.

We started the current period with a prior \( p(x) \) for the location \( x \) of the missile.

We then used the current measurement \( y \) to update to \( p(x \mid y) \).

Finally, we used the law of motion (2.82) for \( \{x_t\} \) to update to \( p_{\text{new}}(x) \).

If we now step into the next period, we are ready to go round again, taking \( p_{\text{new}}(x) \) as the current prior.

Swapping notation \( p_t(x) \) for \( p(x) \) and \( p_{t+1}(x) \) for \( p_{\text{new}}(x) \), the full recursive procedure is:

1. Start the current period with prior \( p_t(x) = N(\hat{x}_t, \Sigma_t) \).
2. Observe current measurement \( y_t \).
3. Compute the filtering distribution \( p_t(x \mid y) = N(\hat{x}_t^F, \Sigma_t^F) \) from \( p_t(x) \) and \( y_t \), applying Bayes rule and the conditional distribution (2.80).
4. Compute the predictive distribution \( p_{t+1}(x) = N(\hat{x}_{t+1}, \Sigma_{t+1}) \) from the filtering distribution and (2.82).
5. Increment \( t \) by one and go to step 1.

Repeating (2.83), the dynamics for \( \hat{x}_t \) and \( \Sigma_t \) are as follows.
\[ \dot{x}_{t+1} = A\dot{x}_t + K_{\Sigma_t}(y_t - G\dot{x}_t) \]
\[ \Sigma_{t+1} = A\Sigma_t A' - K_{\Sigma_t}G\Sigma_t A' + Q \]

These are the standard dynamic equations for the Kalman filter (see, for example, [LS18], page 58)

### 2.7.3 Convergence

The matrix \( \Sigma_t \) is a measure of the uncertainty of our prediction \( \hat{x}_t \) of \( x_t \)

Apart from special cases, this uncertainty will never be fully resolved, regardless of how much time elapses

One reason is that our prediction \( \hat{x}_t \) is made based on information available at \( t - 1 \), not \( t \)

Even if we know the precise value of \( x_{t-1} \) (which we don’t), the transition equation (2.82) implies that \( x_t = Ax_{t-1} + w_t \)

Since the shock \( w_t \) is not observable at \( t - 1 \), any time \( t - 1 \) prediction of \( x_t \) will incur some error (unless \( w_t \) is degenerate)

However, it is certainly possible that \( \Sigma_t \) converges to a constant matrix as \( t \to \infty \)

To study this topic, let’s expand the second equation in (2.83):

\[ \Sigma_{t+1} = A\Sigma_t A' - A\Sigma_t G'(G\Sigma_t G' + R)^{-1}G\Sigma_t A' + Q \] (2.83)

This is a nonlinear difference equation in \( \Sigma_t \)

A fixed point of (2.83) is a constant matrix \( \Sigma \) such that

\[ \Sigma = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q \] (2.84)

Equation (2.83) is known as a discrete time Riccati difference equation

Equation (2.84) is known as a discrete time algebraic Riccati equation

Conditions under which a fixed point exists and the sequence \( \{\Sigma_t\} \) converges to it are discussed in [AHMS96] and [AM05], chapter 4

A sufficient (but not necessary) condition is that all the eigenvalues \( \lambda_i \) of \( A \) satisfy \( |\lambda_i| < 1 \) (cf. e.g., [AM05], p. 77)

(This strong condition assures that the unconditional distribution of \( x_t \) converges as \( t \to +\infty \))

In this case, for any initial choice of \( \Sigma_0 \) that is both nonnegative and symmetric, the sequence \( \{\Sigma_t\} \) in (2.83) converges to a nonnegative symmetric matrix \( \Sigma \) that solves (2.84)
2.7.4 Implementation

The QuantEcon.jl package is able to implement the Kalman filter by using methods for the type Kalman

• Instance data consists of:
  – The parameters $A, G, Q, R$ of a given model
  – the moments $(\hat{x}_t, \Sigma_t)$ of the current prior

• The type Kalman from the QuantEcon.jl package has a number of methods, some that we will wait to use until we study more advanced applications in subsequent lectures

• Methods pertinent for this lecture are:
  – prior_to_filtered, which updates $(\hat{x}_t, \Sigma_t)$ to $(\hat{x}_t^F, \Sigma_t^F)$
  – filtered_to_forecast, which updates the filtering distribution to the predictive distribution – which becomes the new prior $(\hat{x}_{t+1}, \Sigma_{t+1})$
  – update, which combines the last two methods
  – a stationary_values, which computes the solution to (2.84) and the corresponding (stationary) Kalman gain

You can view the program on GitHub

2.7.5 Exercises

Exercise 1

Consider the following simple application of the Kalman filter, loosely based on [LS18], section 2.9.2

Suppose that

• all variables are scalars
• the hidden state $\{x_t\}$ is in fact constant, equal to some $\theta \in \mathbb{R}$ unknown to the modeler

State dynamics are therefore given by (2.82) with $A = 1, Q = 0$ and $x_0 = \theta$

The measurement equation is $y_t = \theta + v_t$ where $v_t$ is $N(0, 1)$ and iid

The task of this exercise to simulate the model and, using the code from kalman.jl, plot the first five predictive densities $p_t(x) = N(\hat{x}_t, \Sigma_t)$

As shown in [LS18], sections 2.9.1–2.9.2, these distributions asymptotically put all mass on the unknown value $\theta$

In the simulation, take $\theta = 10, \hat{x}_0 = 8$ and $\Sigma_0 = 1$

Your figure should – modulo randomness – look something like this
Exercise 2

The preceding figure gives some support to the idea that probability mass converges to $\theta$

To get a better idea, choose a small $\epsilon > 0$ and calculate

$$z_t := 1 - \int_{\theta-\epsilon}^{\theta+\epsilon} p_t(x) dx$$

for $t = 0, 1, 2, \ldots, T$

Plot $z_t$ against $T$, setting $\epsilon = 0.1$ and $T = 600$

Your figure should show error erratically declining something like this
Exercise 3

As discussed above, if the shock sequence \( \{w_t\} \) is not degenerate, then it is not in general possible to predict \( x_t \) without error at time \( t - 1 \) (and this would be the case even if we could observe \( x_{t-1} \))

Let's now compare the prediction \( \hat{x}_t \) made by the Kalman filter against a competitor who is allowed to observe \( x_{t-1} \)

This competitor will use the conditional expectation \( \mathbb{E}[x_t \mid x_{t-1}] \), which in this case is \( Ax_{t-1} \)

The conditional expectation is known to be the optimal prediction method in terms of minimizing mean squared error

(More precisely, the minimizer of \( \mathbb{E} \|x_t - g(x_{t-1})\|^2 \) with respect to \( g \) is \( g^*(x_{t-1}) := \mathbb{E}[x_t \mid x_{t-1}] \))

Thus we are comparing the Kalman filter against a competitor who has more information (in the sense of being able to observe the latent state) and behaves optimally in terms of minimizing squared error

Our horse race will be assessed in terms of squared error

In particular, your task is to generate a graph plotting observations of both \( \|x_t - Ax_{t-1}\|^2 \) and \( \|x_t - \hat{x}_t\|^2 \) against \( t \) for \( t = 1, \ldots, 50 \)

For the parameters, set \( G = I, R = 0.5I \) and \( Q = 0.3I \), where \( I \) is the \( 2 \times 2 \) identity
Set

\[ A = \begin{pmatrix} 0.5 & 0.4 \\ 0.6 & 0.3 \end{pmatrix} \]

To initialize the prior density, set

\[ \Sigma_0 = \begin{pmatrix} 0.9 & 0.3 \\ 0.3 & 0.9 \end{pmatrix} \]

and \( \hat{x}_0 = (8, 8) \)

Finally, set \( x_0 = (0, 0) \)

You should end up with a figure similar to the following (modulo randomness)

Observe how, after an initial learning period, the Kalman filter performs quite well, even relative to the competitor who predicts optimally with knowledge of the latent state

**Exercise 4**

Try varying the coefficient 0.3 in \( Q = 0.3I \) up and down

Observe how the diagonal values in the stationary solution \( \Sigma \) (see (2.84)) 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

### 2.7.6 Solutions

```julia
using QuantEcon

Exercise 1

```import` Distributions: Normal, pdf

# == Parameters == #

\[ \theta = 10 \]
\[ A, G, Q, R = 1.0, 1.0, 0.0, 1.0 \]
\[ x_\text{hat}_0, \Sigma_0 = 8.0, 1.0 \]

# == Initialize Kalman filter == #

kalman = Kalman(A, G, Q, R)
set_state!(kalman, x_\text{hat}_0, \Sigma_0)

# == Run == #

N = 5
xgrid = linspace(\[ \theta - 5, \theta + 2, 200 \])
densities = []
labels = []
for i=1:N
    # Record the current predicted mean and variance, and plot their densities
    m, v = kalman.cur_x_hat, kalman.cur_sigma
    push!(densities, pdf.(Normal(m, sqrt(v)), xgrid))
    push!(labels, LaTeXString("$t = \$i\$"))

    # Generate the noisy signal
    y = \theta + randn()

    # Update the Kalman filter
    update!(kalman, y)
end

plot(xgrid, densities, label=reshape(labels, 1, length(labels)),
    legend=:topleft, grid=false,
    title=LaTeXString("First $N$ densities when $\theta = \theta$"))
```
Exercise 2

```julia
srand(42)  # reproducible results
    = 0.1
kalman = Kalman(A, G, Q, R)
set_state!(kalman, x_hat_0, Σ_0)

nodes, weights = qnwlege(21, θ−, θ+)

T = 600
z = Array(Float64)(T)
for t=1:T
    # Record the current predicted mean and variance, and plot their densities
    m, v = kalman.cur_x_hat, kalman.cur_sigma
    dist = Normal(m, sqrt(v))
    integral = do_quad((x) -> pdf.(dist, x), nodes, weights)
    z[t] = 1. - integral
    # Generate the noisy signal and update the Kalman filter
    update!(kalman, θ + randn())
end
plot(1:T, z, fillrange=0, color=:blue, fillalpha=0.2, grid=false,
    legend=false, xlims=(0, T), ylims=(0, 1))
```

2.7. A First Look at the Kalman Filter
import Distributions: MultivariateNormal, rand
srand(41)  # reproducible results

# === Define A, Q, G, R === #
G = eye(2)
R = 0.5 .* G
A = [0.5 0.4
     0.6 0.3]
Q = 0.3 .* G

# === Define the prior density === #
Σ = [0.9 0.3
    0.3 0.9]
x_hat = [8, 8]''

# === Initialize the Kalman filter === #
kn = Kalman(A, G, Q, R)
set_state!(kn, x_hat, Σ)

# === Set the true initial value of the state === #
x = zeros(2)

# == Print eigenvalues of A == #
println("Eigenvalues of A:
$\$(eigvals(A))")
# == Print stationary Σ == #
S, K = stationary_values(kn)
println("Stationary prediction error variance:
S\n")

# === Generate the plot ===#
T = 50
e1 = Array{Float64}(T)
e2 = Array{Float64}(T)
for t=1:T
    # == Generate signal and update prediction ==#
    y = rand(dist)
    update!(kn, y)

    # == Update state and record error ==#
    Ax = A * x
    x = rand(MultivariateNormal(Ax, Q))
    e1[t] = sum((x - kn.cur_x_hat).^2)
    e2[t] = sum((x - Ax).^2)
end
plot(1:T, e1, color=:black, linewidth=2, alpha=0.6, label="Kalman filter error expectation error")
plot!(1:T, e2, color=:green, linewidth=2, alpha=0.6, label="conditional expectation error")

Eigenvalues of A:
[0.9, -0.1]
Stationary prediction error variance:
[0.403291 0.105072; 0.105072 0.410617]
This section of the course contains foundational models for dynamic economic modeling. Most are single agent problems that take the activities of other agents as given. Later we will look at full equilibrium problems.

3.1 Shortest Paths

Contents

• Shortest Paths
  – Overview
  – Outline of the Problem
  – Finding Least-Cost Paths
  – Solving for J
  – Exercises
  – Solutions

3.1.1 Overview

The shortest path problem is a classic problem in mathematics and computer science with applications in

• Economics (sequential decision making, analysis of social networks, etc.)
• Operations research and transportation
• Robotics and artificial intelligence
• Telecommunication network design and routing
• etc., etc.

Variations of the methods we discuss in this lecture are used millions of times every day, in applications such as
For us, the shortest path problem also provides a nice introduction to the logic of **dynamic programming**. Dynamic programming is an extremely powerful optimization technique that we apply in many lectures on this site.

### 3.1.2 Outline of the Problem

The shortest path problem is one of finding how to traverse a **graph** from one specified node to another at minimum cost.

Consider the following graph:

We wish to travel from node (vertex) $A$ to node $G$ at minimum cost:

- Arrows (edges) indicate the movements we can take.
- Numbers on edges indicate the cost of traveling that edge.

Possible interpretations of the graph include:

- Minimum cost for supplier to reach a destination.
- Routing of packets on the internet (minimize time).
- Etc., etc.

For this simple graph, a quick scan of the edges shows that the optimal paths are:

3.1.3 Finding Least-Cost Paths

For large graphs we need a systematic solution. Let $J(v)$ denote the minimum cost-to-go from node $v$, understood as the total cost from $v$ if we take the best route.
Suppose that we know $J(v)$ for each node $v$, as shown below for the graph from the preceding example.

Note that $J(G) = 0$

The best path can now be found as follows:

- Start at A
- From node $v$, move to any node that solves

$$\min_{w \in F_v} \{c(v, w) + J(w)\} \tag{3.1}$$

where

- $F_v$ is the set of nodes that can be reached from $v$ in one step
- $c(v, w)$ is the cost of traveling from $v$ to $w$

Hence, if we know the function $J$, then finding the best path is almost trivial.

But how to find $J$?

Some thought will convince you that, for every node $v$, the function $J$ satisfies

$$J(v) = \min_{w \in F_v} \{c(v, w) + J(w)\} \tag{3.2}$$

This is known as the Bellman equation, after the mathematician Richard Bellman.
3.1.4 Solving for $J$

The standard algorithm for finding $J$ is to start with

$$J_0(v) = M \text{ if } v \neq \text{destination, else } J_0(v) = 0$$  \hspace{1cm} (3.3)

where $M$ is some large number.

Now we use the following algorithm

1. Set $n = 0$

2. Set $J_{n+1}(v) = \min_{w \in V} \{c(v, w) + J_n(w)\}$ 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.

3.1.5 Exercises

Exercise 1

Use the algorithm given above to find the optimal path (and its cost) for the following graph.

You can put it in a Jupyter notebook cell and hit Shift-Enter it will be saved in the local directory as file `graph.txt`

```%
node0, node1  0.04, node8 11.11, node14 72.21
node1, node46 1247.25, node6 20.59, node13 64.94
node2, node66 54.18, node31 166.80, node45 1561.45
node3, node20 133.65, node6 2.06, node11 42.43
node4, node75 3706.67, node5 0.73, node7 1.02
node5, node45 1382.97, node7 3.33, node11 34.54
node6, node31 63.17, node9 0.72, node10 13.10
node7, node50 478.14, node9 3.15, node10 5.85
node8, node69 577.91, node11 7.45, node12 3.18
node9, node70 2454.28, node13 4.42, node20 16.53
node10, node89 5352.79, node12 1.87, node16 25.16
node11, node94 4961.32, node18 37.55, node20 65.08
node12, node84 3914.62, node24 34.32, node28 170.04
node13, node60 2135.95, node38 236.33, node40 475.33
node14, node67 1878.96, node16 2.70, node24 38.65
node15, node91 3597.11, node17 1.01, node18 2.57
node16, node36 392.92, node19 3.49, node38 278.71
node17, node76 783.29, node22 24.78, node23 26.45
node18, node91 3363.17, node23 16.23, node28 55.84
node19, node26 20.09, node20 0.24, node28 70.54
node20, node98 3523.33, node24 9.81, node33 145.80
node21, node56 626.04, node28 36.65, node31 27.06
node22, node72 1447.22, node39 136.32, node40 124.22
node23, node52 336.73, node26 2.66, node33 22.37
```
| node24, node66 | 875.19 | node26 | 1.80 | node28 | 14.25 |
| node25, node70 | 1343.63 | node32 | 36.58 | node35 | 45.55 |
| node26, node47 | 135.78 | node27 | 0.01 | node42 | 122.00 |
| node27, node65 | 480.55 | node35 | 48.10 | node43 | 246.24 |
| node28, node82 | 2538.18 | node34 | 21.79 | node36 | 15.52 |
| node29, node64 | 635.52 | node32 | 4.22 | node33 | 12.61 |
| node30, node70 | 1343.63 | node35 | 48.10 | node36 | 246.24 |
| node31, node70 | 480.55 | node34 | 21.79 | node36 | 15.52 |
| node32, node82 | 2538.18 | node34 | 21.79 | node36 | 15.52 |

Chapter 3. Dynamic Programming
Writing graph.txt

Here the line node0, node1 0.04, node8 11.11, node14 72.21 means that from node0 we can go to

- node1 at cost 0.04
- node8 at cost 11.11
- node14 at cost 72.21

and so on.

According to our calculations, the optimal path and its cost are like this:

Your code should replicate this result.

### 3.1.6 Solutions

#### Exercise 1

If you Shift-Enter in the next cell you'll save the data we want to work with in the local directory then scroll down for the solution.

```plaintext
f = open("graph.txt", "w")
contents = ""node0, node1 0.04, node8 11.11, node14 72.21
node1, node46 1247.25, node6 20.59, node13 64.94
node2, node66 54.18, node31 166.80, node45 1561.45"
```

3.1. Shortest Paths
node56, node76 197.03, node57 0.02, node61 11.06
node57, node86 701.09, node58 0.46, node60 7.01
node58, node83 556.70, node64 29.85, node65 34.32
node59, node90 820.66, node60 0.72, node71 0.67
node60, node76 48.03, node65 4.76, node67 1.63
node61, node98 1057.59, node63 0.95, node64 4.88
node62, node91 132.23, node64 2.94, node76 38.43
node63, node66 4.43, node72 70.08, node75 56.34
node64, node80 47.73, node65 0.30, node76 11.98
node65, node94 594.93, node66 0.64, node73 33.23
node66, node98 395.63, node68 2.66, node73 37.53
node67, node82 153.53, node68 0.09, node70 0.98
node68, node94 232.10, node70 3.35, node71 1.66
node69, node99 247.80, node70 0.06, node73 8.99
node70, node76 27.18, node72 1.50, node73 8.37
node71, node89 104.50, node74 8.86, node91 284.64
node72, node76 15.32, node84 102.77, node92 133.06
node73, node83 52.22, node76 1.40, node90 243.00
node74, node81 1.07, node76 0.52, node78 8.08
node75, node92 68.53, node76 0.81, node77 1.19
node76, node85 13.18, node77 0.45, node78 2.36
node77, node80 8.94, node78 0.98, node86 64.32
node78, node98 355.90, node81 2.59
node79, node81 0.09, node85 1.45, node91 22.35
node80, node92 121.87, node88 28.78, node98 264.34
node81, node94 99.78, node89 39.52, node92 99.89
node82, node91 47.44, node88 28.05, node93 11.99
node83, node94 114.95, node86 8.75, node88 5.78
node84, node89 19.14, node94 30.41, node98 121.05
node85, node97 94.51, node87 2.66, node89 4.90
node86, node97 85.09
node87, node88 0.21, node91 11.14, node92 21.23
node88, node93 1.31, node91 6.83, node98 6.12
node89, node97 36.97, node99 82.12
node90, node96 23.53, node94 10.47, node99 50.99
node91, node97 22.17
node92, node96 10.83, node97 11.24, node99 34.68
node93, node94 0.19, node97 6.71, node99 32.77
node94, node98 5.91, node96 2.03
node95, node98 6.17, node99 0.27
node96, node98 3.32, node97 0.43, node99 5.87
node97, node98 0.30
node98, node99 0.33
node99,
""
write(f, contents)
close(f)

```julia
function read_graph(in_file)
    graph = Dict()
    infilen = open(in_file, "r")
    for line in readlines(infilen)
        elements = reverse!(split(line, ","))
```

3.1. Shortest Paths 305
node = strip(pop!(elements))
graph[node] = []
if node != "node99"
    for element in elements
        dest, cost = split(element)
        push!(graph[node], [strip(dest), float(cost)])
    end
end
end

close(infile)
return graph
end

function update_J(J, graph)
    next_J = Dict()
    for node in keys(graph)
        if node == "node99"
            next_J[node] = 0
        else
            next_J[node] = minimum([cost + J[dest] for (dest, cost) in graph[node]]))
        end
    end
    return next_J
end

function print_best_path(J, graph)
    sum_costs = 0
    current_location = "node0"
    while current_location != "node99"
        println(current_location)
        running_min = 1e10
        minimizer_dest = "none"
        minimizer_cost = 1e10
        for (dest, cost) in graph[current_location]
            cost_of_path = cost + J[dest]
            if cost_of_path < running_min
                running_min = cost_of_path
                minimizer_cost = cost
                minimizer_dest = dest
            end
        end
        current_location = minimizer_dest
        sum_costs += minimizer_cost
    end
    println("node99")
    @printf \nCost: \$%.2f\n sum_costs
end
graph = read_graph("graph.txt")
M = 1e10
J = Dict()
for node in keys(graph)
    J[node] = M
end
J["node99"] = 0

while true
    next_J = update_J(J, graph)
    if next_J == J
        break
    else
        J = next_J
    end
end

print_best_path(J, graph)

node0
node8
node11
node18
node23
node33
node41
node53
node56
node57
node60
node67
node70
node73
node76
node85
node87
node88
node93
node94
node96
node97
node98
node99

Cost: 160.55

3.2 Job Search I: The McCall Search Model
Questioning a McCall worker is like having a conversation with an out-of-work friend: Maybe you are setting your sights too high, or Why did you quit your old job before you had a new one lined up? This is real social science: an attempt to model, to understand, human behavior by visualizing the situation people find themselves in, the options they face and the pros and cons as they themselves see them. – Robert E. Lucas, Jr.

### 3.2.1 Overview

The McCall search model [McC70] helped transform economists way of thinking about labor markets. To clarify vague notions such as involuntary unemployment, McCall modeled the decision problem of unemployed agents directly, in terms of factors such as

- current and likely future wages
- impatience
- unemployment compensation

To solve the decision problem he used dynamic programming.

Here we set up McCalls model and adopt the same solution method.

As well see, McCalls model is not only interesting in its own right but also an excellent vehicle for learning dynamic programming.

### 3.2.2 The McCall Model

An unemployed worker receives in each period a job offer at wage $W_t$.

At time $t$, our worker has two choices:

1. Accept the offer and work permanently at constant wage $W_t$
2. Reject the offer, receive unemployment compensation $c$, and reconsider next period

The wage sequence $\{W_t\}$ is assumed to be iid with probability mass function $p_1, \ldots, p_n$.

Here $p_i$ is the probability of observing wage offer $W_t = w_i$ in the set $w_1, \ldots, w_n$. 

308 Chapter 3. Dynamic Programming
The worker is infinitely lived and aims to maximize the expected discounted sum of earnings

$$E \sum_{t=0}^{\infty} \beta^t Y_t$$

The constant $\beta$ lies in $(0, 1)$ and is called a **discount factor**

The smaller is $\beta$, the more the worker discounts future utility relative to current utility

The variable $Y_t$ is income, equal to

- his wage $W_t$ when employed
- unemployment compensation $c$ when unemployed

A Trade Off

The worker faces a trade-off:

- Waiting too long for a good offer is costly, since the future is discounted
- Accepting too early is costly, since better offers might arrive in the future

To decide optimally in the face of this trade off, we use dynamic programming

Dynamic programming can be thought of as a two step procedure that

1. first assigns values to states and
2. then deduces optimal actions given those values

Well go through these steps in turn

The Value Function

In order to optimally trade off current and future rewards, we need to think about two things:

1. the current payoffs we get from different choices
2. the different states that those choices will lead to next period (in this case, either employment or unemployment)

To weigh these two aspects of the decision problem, we need to assign *values* to states

To this end, let $V(w)$ be the total lifetime *value* accruing to an unemployed worker who enters the current period unemployed but with wage offer $w$ in hand

More precisely, $V(w)$ denotes the value of the objective function (3.11) when an agent in this situation makes *optimal* decisions now and at all future points in time

Of course $V(w)$ is not trivial to calculate because we dont yet know what decisions are optimal and what arent!

But think of $V$ as a function that assigns to each possible wage $w$ the maximal lifetime value that can be obtained with that offer in hand

3.2. Job Search I: The McCall Search Model
A crucial observation is that this function $V$ must satisfy the recursion

$$
V(w) = \max \left\{ \frac{w}{1 - \beta}, c + \beta \sum_{i=1}^{n} V(w_i)p_i \right\}
$$

for every possible $w_i$ in $w_1, \ldots, w_n$

This important equation is a version of the Bellman equation, which is ubiquitous in economic dynamics and other fields involving planning over time.

The intuition behind it is as follows:

• the first term inside the max operation is the lifetime payoff from accepting current offer $w$, since
  $$w + \beta w + \beta^2 w + \cdots = \frac{w}{1 - \beta}$$

• the second term inside the max operation is the continuation value, which is the lifetime payoff from rejecting the current offer and then behaving optimally in all subsequent periods.

If we optimize and pick the best of these two options, we obtain maximal lifetime value from today, given current offer $w$.

But this is precisely $V(w)$, which is the l.h.s. of (3.4)

### The Optimal Policy

Suppose for now that we are able to solve (3.4) for the unknown function $V$.

Once we have this function in hand we can behave optimally (i.e., make the right choice between accept and reject).

All we have to do is select the maximal choice on the r.h.s. of (3.4).

The optimal action is best thought of as a policy, which is, in general, a map from states to actions.

In our case, the state is the current wage offer $w$.

Given any $w$, we can read off the corresponding best choice (accept or reject) by picking the max on the r.h.s. of (3.4).

Thus, we have a map from $\mathbb{R}$ to $\{0, 1\}$, with 1 meaning accept and zero meaning reject.

We can write the policy as follows

$$
\sigma(w) := 1 \left\{ \frac{w}{1 - \beta} \geq c + \beta \sum_{i=1}^{n} V(w_i)p_i \right\}
$$

Here $1\{P\} = 1$ if statement $P$ is true and equals zero otherwise.

We can also write this as

$$
\sigma(w) := 1\{w \geq \bar{w}\}
$$
where

\[
\tilde{w} := (1 - \beta) \left\{ c + \beta \sum_{i=1}^{n} V(w_i) p_i \right\}
\]

Here \( \tilde{w} \) is a constant depending on \( \beta, c \) and the wage distribution, called the reservation wage.

The agent should accept if and only if the current wage offer exceeds the reservation wage.

Clearly, we can compute this reservation wage if we can compute the value function.

### 3.2.3 Computing the Optimal Policy: Take 1

To put the above ideas into action, we need to compute the value function at points \( w_1, \ldots, w_n \).

In doing so, we can identify these values with the vector \( v = (v_i) \) where \( v_i := V(w_i) \).

In view of (3.4), this vector satisfies the nonlinear system of equations

\[
v_i = \max \left\{ \frac{w_i}{1 - \beta}, c + \beta \sum_{i=1}^{n} v_i p_i \right\} \quad \text{for } i = 1, \ldots, n \tag{3.5}
\]

It turns out that there is exactly one vector \( v := (v_i)_{i=1}^{n} \) in \( \mathbb{R}^n \) that satisfies this equation.

#### The Algorithm

To compute this vector, we proceed as follows:

1. **Step 1**: pick an arbitrary initial guess \( v \in \mathbb{R}^n \)
2. **Step 2**: compute a new vector \( v' \in \mathbb{R}^n \) via

\[
v_i' = \max \left\{ \frac{w_i}{1 - \beta}, c + \beta \sum_{i=1}^{n} v_i p_i \right\} \quad \text{for } i = 1, \ldots, n \tag{3.6}
\]

3. **Step 3**: calculate a measure of the deviation between \( v \) and \( v' \), such as \( \max_i |v_i - v_i'| \)

4. **Step 4**: if the deviation is larger than some fixed tolerance, set \( v = v' \) and go to step 2, else continue

5. **Step 5**: return \( v \)

This algorithm returns an arbitrarily good approximation to the true solution to (3.5), which represents the value function.

(Arbitrarily good means here that the approximation converges to the true solution as the tolerance goes to zero)
The Fixed Point Theory

What's the math behind these ideas?

First, one defines a mapping $T$ from $\mathbb{R}^n$ to itself via

$$
Tv_i = \max \left\{ \frac{w_i}{1 - \beta}, c + \beta \sum_{i=1}^{n} v_i p_i \right\} \quad \text{for } i = 1, \ldots, n
$$

(A new vector $Tv$ is obtained from given vector $v$ by evaluating the r.h.s. at each $i$)

One can show that the conditions of the Banach contraction mapping theorem are satisfied by $T$ as a self-mapping on $\mathbb{R}^n$

One implication is that $T$ has a unique fixed point in $\mathbb{R}^n$

Moreover, its immediate from the definition of $T$ that this fixed point is precisely the value function

The iterative algorithm presented above corresponds to iterating with $T$ from some initial guess $v$

The Banach contraction mapping theorem tells us that this iterative process generates a sequence that converges to the fixed point

Implementation

Let's start with some imports

```julia
using PyPlot
using QuantEcon
using Distributions
using LaTeXStrings
plt = PyPlot
```

Here's the distribution of wage offers we'll work with

```julia
n, a, b = 50, 200, 100
w_min, w_max = 10, 60
const w_vals = linspace(w_min, w_max, n+1)
dist = BetaBinomial(n, a, b)
const p_vals = pdf.(dist, support(dist))
fig, ax = plt.subplots(figsize=(9, 6.5))
ax[{:stem}](w_vals, p_vals, label=L"p_i")
ax[{:set_xlabel}]("wages")
ax[{:set_ylabel}]("probabilities")
plt.show()
```
First let's have a look at the sequence of approximate value functions that the algorithm above generates. Default parameter values are embedded in the function.

Our initial guess $v$ is the value of accepting at every given wage.

```julia
function plot_value_function_seq(ax,
    c=25,
    β=0.99,
    num_plots=6)

    v = w_vals ./ (1 - β)
    v_next = similar(v)

    for i in 1:num_plots
        ax[[:plot]](w_vals, v, label="iterate $i"")

        # Update guess
        for (j, w) in enumerate(w_vals)
            stop_val = w / (1 - β)
            cont_val = c + β * sum(v .* p_vals)
            v_next[j] = max(stop_val, cont_val)
        end
        v[:] = v_next
    end
```

3.2. Job Search I: The McCall Search Model
Here's more serious iteration effort, that continues until measured deviation between successive iterates is below \( tol \)

```julia
function compute_reservation_wage(c::Float64,
    \( \beta \)::Float64;
    max_iter::Int64=500,
    tol::Float64=1e-6)

    # == First compute the value function == #
    v = w_vals ./ (1 - \( \beta \))
    v_next = similar(v)
    i = 0
    error = tol + 1
    while i < max_iter && error > tol
```
for (j, w) in enumerate(w_vals)
  stop_val = w / (1 - \beta)
  cont_val = c + \beta \times \text{sum}(v \times p_vals)
  v_next[j] = \text{max}(stop_val, cont_val)
end

error = \text{maximum}(\text{abs.}(v_{next} - v))
i += 1
v[] = v_{next}  # copy contents into v
end

# == Now compute the reservation wage == #
return (1 - \beta) \times (c + \beta \times \text{sum}(v \times p_vals))
end

Lets compute the reservation wage at the default parameters

c = 25.0
\beta = 0.99
\text{compute\_reservation\_wage}(c, \beta)

47.316499709964695

**Comparative Statics**

Now we know how to compute the reservation wage, lets see how it varies with parameters

In particular, lets look at what happens when we change \beta and c

grid_size = 25
R = \text{rand}((\text{grid\_size}, \text{grid\_size}))
c_vals = \text{linspace}(10.0, 30.0, \text{grid\_size})
\beta_vals = \text{linspace}(0.9, 0.99, \text{grid\_size})

for (i, c) in enumerate(c_vals)
  for (j, \beta) in enumerate(\beta_vals)
    R[i, j] = \text{compute\_reservation\_wage}(c, \beta)
  end
end

fig, ax = \text{plt\_subplots}(\text{figsize=(10, 5.7)})
csl = ax[:,:\text{contourf}](c_vals, \beta_vals, R', \alpha=0.75)
ctr1 = ax[:,:\text{contour}](c_vals, \beta_vals, R')
plt\_clabel ctr1, inline=1, fontsize=13
plt\_colorbar csl, ax=ax

3.2. Job Search I: The McCall Search Model
As expected, the reservation wage increases both with patience and with unemployment compensation.

### 3.2.4 Computing the Optimal Policy: Take 2

The approach to dynamic programming just described is very standard and broadly applicable. For this particular problem, there is also an easier way, which circumvents the need to compute the value function.

Let $\psi$ denote the value of not accepting a job in this period but then behaving optimally in all subsequent periods.

That is,

$$\psi = c + \beta \sum_{i=1}^{n} V(w_i)p_i$$  \hspace{1cm} (3.8)
where $V$ is the value function.

By the Bellman equation, we then have

$$V(w_i) = \max \left\{ \frac{w_i}{1 - \beta}, \psi \right\}$$

Substituting this last equation into (3.8) gives

$$\psi = c + \beta \sum_{i=1}^{n} \max \left\{ \frac{w_i}{1 - \beta}, \psi \right\} p_i$$

(3.9)

This is a nonlinear equation that we can solve for $\psi$.

The natural solution method for this kind of nonlinear equation is iterative. That is,

Step 1: pick an initial guess $\psi$

Step 2: compute the update $\psi'$ via

$$\psi' = c + \beta \sum_{i=1}^{n} \max \left\{ \frac{w_i}{1 - \beta}, \psi \right\} p_i$$

(3.10)

Step 3: calculate the deviation $|\psi - \psi'|$

Step 4: if the deviation is larger than some fixed tolerance, set $\psi = \psi'$ and go to step 2, else continue

Step 5: return $\psi$

Once again, one can use the Banach contraction mapping theorem to show that this process always converges.

The big difference here, however, is that we are iterating on a single number, rather than an $n$-vector.

Here is an implementation:

```julia
function compute_reservation_wage_two(c::Float64; β=0.99, max_iter=500, tol=1e-5)
    # == First compute ψ ==#
    ψ = dot(w_vals, p_vals) ./ (1 - β)
    i = 0
    error = tol + 1
    while i < max_iter && error > tol
        s = max.(w_vals ./ (1 - β), ψ)
        ψ_next = c + β * dot(s, p_vals)
        error = abs.(ψ_next - ψ)
        i += 1
    end
```

3.2. Job Search I: The McCall Search Model 317
You can use this code to solve the exercise below

### 3.2.5 Exercises

**Exercise 1**

Compute the average duration of unemployment when $\beta = 0.99$ and $c$ takes the following values

$$c_{\text{vals}} = \text{np.linspace}(10, 40, 25)$$

That is, start the agent off as unemployed, computed their reservation wage given the parameters, and then simulate to see how long it takes to accept.

Repeat a large number of times and take the average.

Plot mean unemployment duration as a function of $c$ in $c_{\text{vals}}$.

### 3.2.6 Solutions

**Exercise 1**

Here’s one solution

```julia
function compute_stopping_time(w_bar; seed=1234)
    srand(seed)
    stopping_time = 0
    t = 1
    while true
        # Generate a wage draw
        d = DiscreteRV(p_vals)
        w = w_vals[rand(d)]
        if w >= w_bar
            stopping_time = t
            break
        else
            t += 1
        end
    end
    return stopping_time
end
```
function compute_mean_stopping_time(w_bar, num_reps=10000)
    obs = zeros(num_reps)
    for i in 1:num_reps
        obs[i] = compute_stopping_time(w_bar, seed=i)
    end
    return mean(obs)
end

c_vals = linspace(10, 40, 25)
stop_times = similar(c_vals)

for (i, c) in enumerate(c_vals)
    w_bar = compute_reservation_wage_two(c)
    stop_times[i] = compute_mean_stopping_time(w_bar)
end

fig, ax = plt.subplots(figsize=(9, 6.5))

ax[:plot](c_vals, stop_times, label="mean unemployment duration")
ax[:set](xlabel="unemployment compensation", ylabel="months")
ax[:legend]()

plt.show()
3.3 Job Search II: Search and Separation

Contents

- Job Search II: Search and Separation
  - Overview
  - The Model
  - Solving the Model using Dynamic Programming
  - Implementation
  - The Reservation Wage
  - Exercises
  - Solutions
3.3.1 Overview

Previously we looked at the McCall job search model \cite{McC70} as a way of understanding unemployment and worker decisions.

One unrealistic feature of the model is that every job is permanent.

In this lecture we extend the McCall model by introducing job separation.

Once separation enters the picture, the agent comes to view:

- the loss of a job as a capital loss, and
- a spell of unemployment as an investment in searching for an acceptable job.

3.3.2 The Model

The model concerns the life of an infinitely lived worker and

- the opportunities he or she (let’s say he to save one character) has to work at different wages
- exogenous events that destroy his current job
- his decision making process while unemployed.

The worker can be in one of two states: employed or unemployed.

He wants to maximize

$$E \sum_{t=0}^{\infty} \beta^t u(Y_t)$$  \hfill (3.11)

The only difference from the baseline model is that we’ve added some flexibility over preferences by introducing a utility function \( u \).

It satisfies \( u' > 0 \) and \( u'' < 0 \).

Timing and Decisions

Here’s what happens at the start of a given period in our model with search and separation.

If currently employed, the worker consumes his wage \( w \), receiving utility \( u(w) \).

If currently unemployed, he

- receives and consumes unemployment compensation \( c \)
- receives an offer to start work next period at a wage \( w' \) drawn from a known distribution \( p_1, \ldots, p_n \).

He can either accept or reject the offer.

If he accepts the offer, he enters next period employed with wage \( w' \).

If he rejects the offer, he enters next period unemployed.
When employed, the agent faces a constant probability $\alpha$ of becoming unemployed at the end of the period (Note: we do not allow for job search while employed this topic is taken up in a later lecture)

### 3.3.3 Solving the Model using Dynamic Programming

Let

- $V(w)$ be the total lifetime value accruing to a worker who enters the current period employed with wage $w$
- $U$ be the total lifetime value accruing to a worker who is unemployed this period

Here value means the value of the objective function (3.11) when the worker makes optimal decisions at all future points in time.

Suppose for now that the worker can calculate the function $V$ and the constant $U$ and use them in his decision making.

Then $V$ and $U$ should satisfy

$$V(w) = u(w) + \beta[(1 - \alpha)V(w) + \alpha U] \quad (3.12)$$

and

$$U = u(c) + \beta \sum_i \max \{U, V(w_i)\} p_i \quad (3.13)$$

Let’s interpret these two equations in light of the fact that today’s tomorrow is tomorrow’s today

- The left hand sides of equations (3.12) and (3.13) are the values of a worker in a particular situation today.
- The right hand sides of the equations are the discounted (by $\beta$) expected values of the possible situations that worker can be in tomorrow.
- But tomorrow the worker can be in only one of the situations whose values today are on the left sides of our two equations.

Equation (3.13) incorporates the fact that a currently unemployed worker will maximize his own welfare.

In particular, if his next period wage offer is $w'$, he will choose to remain unemployed unless $U < V(w')$.

Equations (3.12) and (3.13) are the Bellman equations for this model.

Equations (3.12) and (3.13) provide enough information to solve out for both $V$ and $U$.

Before discussing this, however, let’s make a small extension to the model.
Stochastic Offers

Let’s suppose now that unemployed workers don’t always receive job offers.

Instead, let’s suppose that unemployed workers only receive an offer with probability $\gamma$.

If our worker does receive an offer, the wage offer is drawn from $p$ as before.

He either accepts or rejects the offer.

Otherwise the model is the same.

With some thought, you will be able to convince yourself that $V$ and $U$ should now satisfy

\[ V(w) = u(w) + \beta[(1 - \alpha)V(w) + \alpha U] \] (3.14)

and

\[ U = u(c) + \beta(1 - \gamma)U + \beta\gamma \sum_i \max\{U, V(w_i)\} p_i \] (3.15)

Solving the Bellman Equations

We’ll use the same iterative approach to solving the Bellman equations that we adopted in the first job search lecture.

Here this amounts to

1. make guesses for $U$ and $V$
2. plug these guesses into the right hand sides of (3.14) and (3.15)
3. update the left hand sides from this rule and then repeat

In other words, we are iterating using the rules

\[ V_{n+1}(w_i) = u(w_i) + \beta[(1 - \alpha)V_n(w_i) + \alpha U_n] \] (3.16)

and

\[ U_{n+1} = u(c) + \beta(1 - \gamma)U_n + \beta\gamma \sum_i \max\{U_n, V_n(w_i)\} p_i \] (3.17)

starting from some initial conditions $U_0, V_0$.

As before, the system always converges to the true solutions in this case, the $V$ and $U$ that solve (3.14) and (3.15).

A proof can be obtained via the Banach contraction mapping theorem.
3.3.4 Implementation

Lets implement this iterative process

In the code you'll see that we use a type to store the various parameters and other objects associated with a given model

This helps to tidy up the code and provides an object that's easy to pass to functions

The default utility function is a CRRA utility function

```Julia
using Distributions

# A default utility function

function u(c::Real, σ::Real)
    if c > 0
        return (c^(1 - σ) - 1) / (1 - σ)
    else
        return -10e6
    end
end

# default wage vector with probabilities

const n = 60  # n possible outcomes for wage
const default_w_vec = linspace(10, 20, n)  # wages between 10 and 20
const a, b = 600, 400  # shape parameters
const dist = BetaBinomial(n-1, a, b)
const default_p_vec = pdf.(dist, support(dist))

mutable struct McCallModel{TF <: AbstractFloat, TAV <: AbstractVector{TF}, TAV2 <: AbstractVector{TF}}
    α::TF  # Job separation rate
    β::TF  # Discount rate
    γ::TF  # Job offer rate
    c::TF  # Unemployment compensation
    σ::TF  # Utility parameter
    w_vec::TAV  # Possible wage values
    p_vec::TAV2  # Probabilities over w_vec

    McCallModel(α::TF=0.2,
                β::TF=0.98,
                γ::TF=0.7,
                c::TF=6.0,
                σ::TF=2.0,
                w_vec::TAV=default_w_vec,
                p_vec::TAV2=default_p_vec) where {TF, TAV, TAV2} =
        new(TF, TAV, TAV2)(α, β, γ, c, σ, w_vec, p_vec)
end

""
A function to update the Bellman equations. Note that V_new is modified in
```
The approach is to iterate until successive iterates are closer together than some small tolerance level

We then return the current iterate as an approximate solution

Let's plot the approximate solutions $U$ and $V$ to see what they look like

Well use the default parameterizations found in the code above
using Plots, LaTeXStrings
pyplot()

mcm = McCallModel()
V, U = solve_mccall_model(mcm)
U_vec = U .* ones(length(mcm.w_vec))

plot(mcm.w_vec,
    [V U_vec],
    lw=2,
    α=0.7,
    label=L"V" L"U")

Heres the plot this code produces

![Plot](image)

The value $V$ is increasing because higher $w$ generates a higher wage flow conditional on staying employed.

### 3.3.5 The Reservation Wage

Once $V$ and $U$ are known, the agent can use them to make decisions in the face of a given wage offer.

If $V(w) > U$, then working at wage $w$ is preferred to unemployment.

If $V(w) < U$, then remaining unemployed will generate greater lifetime value.

Suppose in particular that $V$ crosses $U$ (as it does in the preceding figure).
Then, since $V$ is increasing, there is a unique smallest $w$ in the set of possible wages such that $V(w) \geq U$.

We denote this wage $\bar{w}$ and call it the reservation wage.

Optimal behavior for the worker is characterized by $\bar{w}$

- if the wage offer $w$ in hand is greater than or equal to $\bar{w}$, then the worker accepts
- if the wage offer $w$ in hand is less than $\bar{w}$, then the worker rejects

Here is a function called `compute_reservation_wage` that takes an instance of a McCall model and returns the reservation wage associated with a given model.

It uses `numpy.searchsorted` to obtain the first $w$ in the set of possible wages such that $V(w) > U$.

If $V(w) < U$ for all $w$, then the function returns `numpy.inf`.

```python
""
Computes the reservation wage of an instance of the McCall model by finding the smallest w such that V(w) > U.

If V(w) > U for all w, then the reservation wage w_bar is set to the lowest wage in mcm.w_vec.

If V(w) < U for all w, then w_bar is set to numpy.inf.

Parameters
---------
mcm : an instance of McCallModel
return_values : bool (optional, default=false)
    Return the value functions as well

Returns
-------
w_bar : scalar
    The reservation wage
""

function compute_reservation_wage(mcm::McCallModel; return_values::Bool=false)

    V, U = solve_mccall_model(mcm)
    w_idx = searchsortedfirst(V - U, 0)

    if w_idx == length(V)
        w_bar = Inf
    else
        w_bar = mcm.w_vec[w_idx]
    end

    if return_values == false
        return w_bar
    else
        return w_bar, V, U
    end

end
```

3.3. Job Search II: Search and Separation
Lets use it to look at how the reservation wage varies with parameters.

In each instance below, we will show you a figure and then ask you to reproduce it in the exercises.

**The Reservation Wage and Unemployment Compensation**

First, let’s look at how \( \bar{w} \) varies with unemployment compensation.

In the figure below, we use the default parameters in the `McCallModel` type, apart from \( c \) (which takes the values given on the horizontal axis).

As expected, higher unemployment compensation causes the worker to hold out for higher wages. In effect, the cost of continuing job search is reduced.

**The Reservation Wage and Discounting**

Next, let’s investigate how \( \bar{w} \) varies with the discount rate.

The next figure plots the reservation wage associated with different values of \( \beta \).
Again, the results are intuitive: More patient workers will hold out for higher wages.

**The Reservation Wage and Job Destruction**

Finally, let's look at how $\tilde{w}$ varies with the job separation rate $\alpha$.

Higher $\alpha$ translates to a greater chance that a worker will face termination in each period once employed.
Once more, the results are in line with our intuition.
If the separation rate is high, then the benefit of holding out for a higher wage falls.
Hence the reservation wage is lower.

### 3.3.6 Exercises

**Exercise 1**

Reproduce all the reservation wage figures shown above.

**Exercise 2**

Plot the reservation wage against the job offer rate $\gamma$.

Use

```plaintext
grid_size = 25
γ_vals = linspace(0.05, 0.95, grid_size)
```

Interpret your results.
3.3.7 Solutions

Exercise 1

Using the `compute_reservation_wage` function mentioned earlier in the lecture, we can create an array for reservation wages for different values of $c$, $\beta$ and $\alpha$ and plot the results like so:

```julia
grid_size = 25
c_vals = linspace(2, 12, grid_size)
w_bar_vals = similar(c_vals)

mcm = McCallModel()

for (i, c) in enumerate(c_vals)
    mcm.c = c
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar
end

plot(c_vals, w_bar_vals, lw=2, alpha=0.7, xlabel="unemployment compensation", ylabel="reservation wage", label=L"\bar\omega as a function of $c$")
```

Exercise 2

Similar to above, we can plot $\bar{\omega}$ against $\gamma$ as follows:

```julia
grid_size = 25
γ_vals = linspace(0.05, 0.95, grid_size)
w_bar_vals = similar(γ_vals)

mcm = McCallModel()

for (i, γ) in enumerate(γ_vals)
    mcm.γ = γ
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar
end

plot(γ_vals, w_bar_vals, lw=2, alpha=0.7, xlabel="job offer rate", ylabel="reservation wage", label=L"\bar\omega as a function of $\gamma$")
```
As expected, the reservation wage increases in $\gamma$
This is because higher $\gamma$ translates to a more favorable job search environment
Hence workers are less willing to accept lower offers

### 3.4 A Problem that Stumped Milton Friedman

(and that Abraham Wald solved by inventing sequential analysis)
3.4.1 Overview

This lecture describes a statistical decision problem encountered by Milton Friedman and W. Allen Wallis during World War II when they were analysts at the U.S. Governments Statistical Research Group at Columbia University.

This problem led Abraham Wald [Wal47] to formulate sequential analysis, an approach to statistical decision problems intimately related to dynamic programming.

In this lecture, we apply dynamic programming algorithms to Friedman and Wallis and Wald’s problem.

Key ideas in play will be:

- Bayes Law
- Dynamic programming
- Type I and type II statistical errors
  - a type I error occurs when you reject a null hypothesis that is true
  - a type II error is when you accept a null hypothesis that is false
- Abraham Wald’s sequential probability ratio test
- The power of a statistical test
- The critical region of a statistical test
- A uniformly most powerful test

3.4.2 Origin of the problem

On pages 137-139 of his 1998 book Two Lucky People with Rose Friedman [FF98], Milton Friedman described a problem presented to him and Allen Wallis during World War II, when they worked at the US Governments Statistical Research Group at Columbia University.

Let’s listen to Milton Friedman tell us what happened.

In order to understand the story, it is necessary to have an idea of a simple statistical problem, and of the standard procedure for dealing with it. The actual problem out of which sequential analysis grew will serve. The Navy has two alternative designs (say A and B) for a projectile. It wants to determine which is superior.

To do so it undertakes a series of paired firings. On each round it assigns the value 1 or 0 to A accordingly as its performance is superior or inferior to that of B and conversely 0 or 1 to B. The Navy asks the statistician how to conduct the test and how to analyze the results.

The standard statistical answer was to specify a number of firings (say 1,000) and a pair of percentages (e.g., 53% and 47%) and tell the client that if A receives a 1 in more than 53% of the firings, it can be regarded as superior; if it receives a 1 in fewer than 47%, B can be regarded as superior; if the percentage is between 47% and 53%, neither can be so regarded.

When Allen Wallis was discussing such a problem with (Navy) Captain Garret L. Schyler, the captain objected that such a test, to quote from Allen’s account, may prove wasteful. If a wise and seasoned ordnance officer like Schyler were on the premises, he would see after the first few thousand or even few hundred firings that the test is premature.
[rounds] that the experiment need not be completed either because the new method is obviously inferior or because it is obviously superior beyond what was hoped for . . .

Friedman and Wallis struggled with the problem but, after realizing that they were not able to solve it, described the problem to Abraham Wald

That started Wald on the path that led him to Sequential Analysis [Wal47]

Well formulate the problem using dynamic programming

### 3.4.3 A dynamic programming approach

The following presentation of the problem closely follows Dmitri Berskekass treatment in Dynamic Programming and Stochastic Control [Ber75]

A decision maker observes iid draws of a random variable $z$

He (or she) wants to know which of two probability distributions $f_0$ or $f_1$ governs $z$

After a number of draws, also to be determined, he makes a decision as to which of the distributions is generating the draws he observers

To help formalize the problem, let $x \in \{x_0, x_1\}$ be a hidden state that indexes the two distributions:

$$
P\{z = v \mid x\} = \begin{cases} f_0(v) & \text{if } x = x_0, \\ f_1(v) & \text{if } x = x_1 \end{cases}
$$

Before observing any outcomes, the decision maker believes that the probability that $x = x_0$ is

$$
p_{-1} = P\{x = x_0 \mid \text{no observations}\} \in (0, 1)
$$

After observing $k + 1$ observations $z_k, z_{k-1}, \ldots, z_0$, he updates this value to

$$
p_k = P\{x = x_0 \mid z_k, z_{k-1}, \ldots, z_0\},
$$

which is calculated recursively by applying Bayes law:

$$
p_{k+1} = \frac{p_k f_0(z_{k+1})}{p_k f_0(z_{k+1}) + (1 - p_k) f_1(z_{k+1})}, \quad k = -1, 0, 1, \ldots
$$

After observing $z_k, z_{k-1}, \ldots, z_0$, the decision maker believes that $z_{k+1}$ has probability distribution

$$
f(v) = p_k f_0(v) + (1 - p_k) f_1(v)
$$

This is a mixture of distributions $f_0$ and $f_1$, with the weight on $f_0$ being the posterior probability that $x = x_0$.

To help illustrate this kind of distribution, lets inspect some mixtures of beta distributions

---

1 Because the decision maker believes that $z_{k+1}$ is drawn from a mixture of two i.i.d. distributions, he does not believe that the sequence $[z_{k+1}, z_{k+2}, \ldots]$ is i.i.d. Instead, he believes that it is exchangeable. See [Kre88] chapter 11, for a discussion of exchangeability.
The density of a beta probability distribution with parameters $a$ and $b$ is

$$f(z; a, b) = \frac{\Gamma(a + b) z^{a-1}(1 - z)^{b-1}}{\Gamma(a) \Gamma(b)} \quad \text{where} \quad \Gamma(t) := \int_0^\infty x^{t-1} e^{-x} dx$$

Well discretize this distribution to make it more straightforward to work with.

The next figure shows two discretized beta distributions in the top panel.

The bottom panel presents mixtures of these distributions, with various mixing probabilities $p_k$.

```julia
using Distributions
using Plots
using LaTeXStrings

f0 = pdf(Beta(1, 1), linspace(0, 1))
f0 = f0 / sum(f0)
f1 = pdf(Beta(9, 9), linspace(0, 1))
f1 = f1 / sum(f1) # Make sure sums to 1

a = plot([f0 f1],
    xlabel=L"$k$ Values",
ylabel=L"Probability of $z_k$",
    labels=[L"f_0" L"f_1"],
    linewidth=2,
    ylims=[0.; 0.07],
    title="Original Distributions")

mix = Array{Float64}(50, 3)
labels = Array{String}(1, 3)
p_k = [0.25; 0.5; 0.75]
for i in 1:3
    mix[:, i] = p_k[i] * f0 + (1 - p_k[i]) * f1
    labels[1, i] = string(L"p_k = ", p_k[i])
end

b = plot(mix,
    xlabel=L"$k$ Values",
ylabel=L"Probability of $z_k$",
    labels=labels,
    linewidth=2,
    ylims=[0.; 0.06],
    title="Mixture of Original Distributions")

plot(a, b, layout=(2, 1), size=(800, 600))
```
Losses and costs

After observing \( z_k, z_{k-1}, \ldots, z_0 \), the decision maker chooses among three distinct actions:

- He decides that \( x = x_0 \) and draws no more \( z \)s
- He decides that \( x = x_1 \) and draws no more \( z \)s
- He postpones deciding now and instead chooses to draw a \( z_{k+1} \)

Associated with these three actions, the decision maker can suffer three kinds of losses:

- A loss \( L_0 \) if he decides \( x = x_0 \) when actually \( x = x_1 \)
- A loss \( L_1 \) if he decides \( x = x_1 \) when actually \( x = x_0 \)
- A cost \( c \) if he postpones deciding and chooses instead to draw another \( z \)
Digression on type I and type II errors

If we regard $x = x_0$ as a null hypothesis and $x = x_1$ as an alternative hypothesis, then $L_1$ and $L_0$ are losses associated with two types of statistical errors.

- a type I error is an incorrect rejection of a true null hypothesis (a false positive)
- a type II error is a failure to reject a false null hypothesis (a false negative)

So when we treat $x = x_0$ as the null hypothesis

- We can think of $L_1$ as the loss associated with a type I error
- We can think of $L_0$ as the loss associated with a type II error

Intuition

Let's try to guess what an optimal decision rule might look like before we go further

Suppose at some given point in time that $p$ is close to 1

Then our prior beliefs and the evidence so far point strongly to $x = x_0$

If, on the other hand, $p$ is close to 0, then $x = x_1$ is strongly favored

Finally, if $p$ is in the middle of the interval $[0, 1]$, then we have little information in either direction

This reasoning suggests a decision rule such as the one shown in the figure

As well see, this is indeed the correct form of the decision rule

The key problem is to determine the threshold values $\alpha, \beta$, which will depend on the parameters listed above

You might like to pause at this point and try to predict the impact of a parameter such as $c$ or $L_0$ on $\alpha$ or $\beta$

A Bellman equation

Let $J(p)$ be the total loss for a decision maker with current belief $p$ who chooses optimally

With some thought, you will agree that $J$ should satisfy the Bellman equation

$$J(p) = \min \{(1 - p)L_0, \ pL_1, \ c + \mathbb{E}[J(p')]\}$$

(3.18)
where $p'$ is the random variable defined by

$$p' = \frac{pf_0(z)}{pf_0(z) + (1 - p)f_1(z)}$$

when $p$ is fixed and $z$ is drawn from the current best guess, which is the distribution $f$ defined by

$$f(v) = pf_0(v) + (1 - p)f_1(v)$$

In the Bellman equation, minimization is over three actions:

1. accept $x_0$
2. accept $x_1$
3. postpone deciding and draw again

Let

$$A(p) := \mathbb{E}[J(p')]$$

Then we can represent the Bellman equation as

$$J(p) = \min \{ (1 - p)L_0, \ pL_1, \ c + A(p) \}$$

where $p \in [0, 1]$

Here

- $(1 - p)L_0$ is the expected loss associated with accepting $x_0$ (i.e., the cost of making a type II error)
- $pL_1$ is the expected loss associated with accepting $x_1$ (i.e., the cost of making a type I error)
- $c + A(p)$ is the expected cost associated with drawing one more $z$

The optimal decision rule is characterized by two numbers $\alpha, \beta \in (0, 1) \times (0, 1)$ that satisfy

$$(1 - p)L_0 < \min\{pL_1, c + A(p)\} \text{ if } p \geq \alpha$$

and

$$pL_1 < \min\{(1 - p)L_0, c + A(p)\} \text{ if } p \leq \beta$$

The optimal decision rule is then

$$\begin{align*}
\text{accept } x = x_0 & \text{ if } p \geq \alpha \\
\text{accept } x = x_1 & \text{ if } p \leq \beta \\
\text{draw another } z & \text{ if } \beta \leq p \leq \alpha
\end{align*}$$

Our aim is to compute the value function $J$, and from it the associated cutoffs $\alpha$ and $\beta$

One sensible approach is to write the three components of $J$ that appear on the right side of the Bellman equation as separate functions

Later, doing this will help us obey the dont repeat yourself (DRY) golden rule of coding
3.4.4 Implementation

Let's code this problem up and solve it.

To approximate the value function that solves Bellman equation (3.18), we use value function iteration

- For earlier examples of this technique see the shortest path, job search or optimal growth lectures

As in the optimal growth lecture, to approximate a continuous value function

- We iterate at a finite grid of possible values of \( p \)
- When we evaluate \( A(p) \) between grid points, we use linear interpolation

This means that to evaluate \( J(p) \) where \( p \) is not a grid point, we must use two points:

- First, we use the largest of all the grid points smaller than \( p \), and call it \( p_i \)
- Second, we use the grid point immediately after \( p \), named \( p_{i+1} \), to approximate the function value as

\[
J(p) = J(p_i) + (p - p_i)\frac{J(p_{i+1}) - J(p_i)}{p_{i+1} - p_i}
\]

In one dimension, you can think of this as simply drawing a line between each pair of points on the grid.

Here's the code.

```julia
using QuantEcon.compute_fixed_point, QuantEcon.DiscreteRV, QuantEcon.draw, QuantEcon.LinInterp

# For a given probability return expected loss of choosing model 0
expect_loss_choose_0(p::Real, L0::Real) = (1 - p) * L0

# For a given probability return expected loss of choosing model 1
expect_loss_choose_1(p::Real, L1::Real) = p * L1

# We will need to be able to evaluate the expectation of our Bellman equation \( J \). In order to do this, we need the current probability that model 0 is correct \( p \), the distributions \( (f_0, f_1) \), and a function that can evaluate the Bellman equation

function EJ(p::Real, f0::AbstractVector, f1::AbstractVector, J::LinInterp)
    # Get the current distribution we believe \( (p * f_0 + (1 - p) * f_1) \)
    curr_dist = p * f0 + (1 - p) * f1

    # Get tomorrow's expected distribution through Bayes law
    tp1_dist = clamp((p * f0) ./ (p * f0 + (1 - p) * f1), 0, 1)

    # Evaluate the expectation
    EJ = dot(curr_dist, J.(tp1_dist))

    return EJ
```

3.4. A Problem that Stumped Milton Friedman 339
end

expect_loss_cont(p::Real, c::Real,
    f0::AbstractVector, f1::AbstractVector, J::LinInterp) =
    c + EJ(p, f0, f1, J)

""
Evaluates the value function for a given continuation value
function; that is, evaluates

J(p) = min(p L0, (1-p)L1, c + E[J(p')])
""

function bellman_operator(pgrid::AbstractVector,
    c::Real,
    f0::AbstractVector,
    f1::AbstractVector,
    L0::Real,
    L1::Real,
    J::AbstractVector)

    m = length(pgrid)
    @assert m == length(J)

    J_out = zeros(m)
    J_interp = LinInterp(pgrid, J)

    for (p_ind, p) in enumerate(pgrid)
        # Payoff of choosing model 0
        p_c_0 = expect_loss_choose_0(p, L0)
        p_c_1 = expect_loss_choose_1(p, L1)
        p_con = expect_loss_cont(p, c, f0, f1, J_interp)

        J_out[p_ind] = min(p_c_0, p_c_1, p_con)
    end

    return J_out
end

# Create two distributions over 50 values for k
# We are using a discretized beta distribution

p_m1 = linspace(0, 1, 50)
f0 = clamp.(pdf.(Beta(1, 1), p_m1), 1e-8, Inf)
f0 = f0 / sum(f0)
f1 = clamp.(pdf.(Beta(9, 9), p_m1), 1e-8, Inf)
f1 = f1 / sum(f1)

# To solve
pg = linspace(0, 1, 251)
J1 = compute_fixed_point(x -> bellman_operator(pg, 0.5, f0, f1, 5.0, 5.0, x), zeros(length(pg)), err_tol=1e-6, print_skip=5);
Running it produces the following output on our machine

Compute iterate 5 with error 0.08552607733051265
Compute iterate 10 with error 0.00038782894418165625
Compute iterate 15 with error 1.6097835344730527e-6
Converged in 16 steps

The distance column shows the maximal distance between successive iterates.
This converges to zero quickly, indicating a successful iterative procedure.
Iteration terminates when the distance falls below some threshold.

**A more sophisticated implementation**

Now for some gentle criticisms of the preceding code.

By writing the code in terms of functions, we have to pass around some things that are constant throughout the problem:

- \( c, f_0, f_1, L_0, \) and \( L_1 \)

So now let's turn our simple script into a type.
This will allow us to simplify the function calls and make the code more reusable.

We shall construct two types that:

- store all of our parameters for us internally
- represent the solution to our Bellman equation alongside the \( \alpha \) and \( \beta \) decision cutoffs
- accompany many of the same functions used above which now act on the type directly
- allow us, in addition, to simulate draws and the decision process under different prior beliefs.

```julia
#=
Author: Shunsuke Hori
=
""
""""This type is used to store the solution to the problem presented in the "Wald Friedman" notebook presented on the QuantEcon website.

Solution
--------
J : AbstractVector
    Discretized value function that solves the Bellman equation
lb : Real
    Lower cutoff for continuation decision
ub : Real
    Upper cutoff for continuation decision
""""
```
mutable struct WFSolution{TAV <: AbstractVector, TR::Real}
    J::TAV
    lb::TR
    ub::TR
end

""
This type is used to solve the problem presented in the "Wald Friedman"
notebook presented on the QuantEcon website.
Parameters
--------
c : Real
    Cost of postponing decision
L0 : Real
    Cost of choosing model 0 when the truth is model 1
L1 : Real
    Cost of choosing model 1 when the truth is model 0
f0 : AbstractVector
    A finite state probability distribution
f1 : AbstractVector
    A finite state probability distribution
m : Integer
    Number of points to use in function approximation
""
struct WaldFriedman{TR :: Real,
    TI :: Integer,
    TAV1 :: AbstractVector,
    TAV2 :: AbstractVector)
    c::TR
    L0::TR
    L1::TR
    f0::TAV1
    f1::TAV1
    m::TI
    pgrid::TAV2
    sol::WFSolution
end

function WaldFriedman(c::Real,
    L0::Real,
    L1::Real,
    f0::AbstractVector,
    f1::AbstractVector;
    m::Integer=25)
    pgrid = linspace(0.0, 1.0, m)
    # Renormalize distributions so nothing is "too" small
    f0 = clamp.(f0, 1e-8, 1-1e-8)
    f1 = clamp.(f1, 1e-8, 1-1e-8)
    f0 = f0 / sum(f0)
    f1 = f1 / sum(f1)
J = zeros(m)
lb = 0.
ub = 0.

WaldFriedman(c, L0, L1, f0, f1, m, pgrid, WFSolution(J, lb, ub))

end

""
This function takes a value for the probability with which
the correct model is model 0 and returns the mixed
distribution that corresponds with that belief.
""
current_distribution(wf::WaldFriedman, p::Real) = p * wf.f0 + (1 - p) * wf.f1

""
This function takes a value for p, and a realization of the
random variable and calculates the value for p tomorrow.
""
function bayes_update_k(wf::WaldFriedman, p::Real, k::Integer)
f0_k = wf.f0[k]
f1_k = wf.f1[k]

p_t0p1 = p * f0_k / (p * f0_k + (1 - p) * f1_k)

return clamp(p_t0p1, 0, 1)

end

""
This is similar to `bayes_update_k` except it returns a
new value for p for each realization of the random variable
""
bayes_update_all(wf::WaldFriedman, p::Real) =
    clamp.(p * wf.f0 ./(p * wf.f0 + (1 - p) * wf.f1), 0, 1)

""
For a given probability specify the cost of accepting model 0
""
payoff_choose_f0(wf::WaldFriedman, p::Real) = (1 - p) * wf.L0

""
For a given probability specify the cost of accepting model 1
""
payoff_choose_f1(wf::WaldFriedman, p::Real) = p * wf.L1

""
This function evaluates the expectation of the value function
at period t+1. It does so by taking the current probability
distribution over outcomes:

    p(z_{k+1}) = p_k f_0(z_{k+1}) + (1-p_k) f_1(z_{k+1})

and evaluating the value function at the possible states
tomorrow J(p_{t+1}) where

3.4. A Problem that Stumped Milton Friedman
\[ p_{(t+1)} = p \frac{f_0}{(p f_0 + (1-p) f_1)} \]

Parameters
---------
p : Real
  The current believed probability that model 0 is the true
  model.
J : LinInterp
  The current value function for a decision to continue

Returns
-------
EJ : scalar
  The expected value of the value function tomorrow

```
function EJ(wf::WaldFriedman, p::Real, J::LinInterp)
  # Pull out information
  f0, f1 = wf.f0, wf.f1

  # Get the current believed distribution and tomorrows possible dists
  # Need to clip to make sure things don't blow up (go to infinity)
  curr_dist = current_distribution(wf, p)
  tpl1_dist = bayes_update_all(wf, p)

  # Evaluate the expectation
  EJ = dot(curr_dist, J.(tpl1_dist))

  return EJ
end
```

For a given probability distribution and value function give
cost of continuing the search for correct model
```
payoff_continue(wf::WaldFriedman, p::Real, J::LinInterp) = wf.c + EJ(wf, p, J)
```

Evaluates the value function for a given continuation value
function; that is, evaluates

\[ J(p) = \min( (1-p)L0, pL1, c + E[J(p')]) \]

Uses linear interpolation between points
```
function bellman_operator(wf::WaldFriedman, J::AbstractVector)
  c, L0, L1, f0, f1 = wf.c, wf.L0, wf.L1, wf.f0, wf.f1
  m, pgrid = wf.m, wf.pgrid

  J_out = similar(J)
  J_interp = LinInterp(pgrid, J)

  for (p_ind, p) in enumerate(pgrid)
    J_out[p_ind] = J_interp(p)
  end

  return J_out
end
```
# Payoff of choosing model 0
\[ p_{c_0} = \text{payoff\_choose\_f0}(wf, p) \]
\[ p_{c_1} = \text{payoff\_choose\_f1}(wf, p) \]
\[ p_{\text{con}} = \text{payoff\_continue}(wf, p, J_{\text{interp}}) \]
\[ J_{\text{out}}[p_{\text{ind}}] = \min(p_{c_0}, p_{c_1}, p_{\text{con}}) \]
end
return \( J_{\text{out}} \)
end

function find_cutoff_rule(wf::WaldFriedman, J::AbstractVector)
  \( m, pgrid = wf.m, wf.pgrid \)

  # Evaluate cost at all points on grid for choosing a model
  \( p_{c_0} = \text{payoff\_choose\_f0}(wf, pgrid) \)
  \( p_{c_1} = \text{payoff\_choose\_f1}(wf, pgrid) \)

  # The cutoff points can be found by differencing these costs with
  # the Bellman equation (\( J \) is always less than or equal to \( p_{c_i} \))
  lb = pgrid[searchsortedlast(p_{c_1} - J, 1e-10)]
  ub = pgrid[searchsortedlast(J - p_{c_0}, -1e-10)]

  return lb, ub
end

function solve_model!(wf::WaldFriedman; tol::AbstractFloat=1e-7)
  bell_op(x) = bellman_operator(wf, x)
  J = \text{compute\_fixed\_point}(bell_op, zeros(wf.m), err\_tol=tol, print\_skip=5)

  wf.sol.J = J
  wf.sol.lb, wf.sol.ub = find_cutoff_rule(wf, J)

  return J
end

function simulate(wf::WaldFriedman, f::AbstractVector; p0::Real=0.5)
  # Check whether \( vf \) is computed
  if sum(abs, wf.sol.J) < 1e-8
    solve_model!(wf)
  end

  # Unpack useful info
  lb, ub = wf.sol.lb, wf.sol.ub
drv = \text{DiscreteRV}(f)
# Initialize a couple useful variables
decision = 0
p = p0
t = 0

while true
    # Maybe should specify which distribution is correct one so that
    # the draws come from the "right" distribution
    k = rand(drv)
    t = t + 1
    p = bayes_update_k(wf, p, k)
    if p < lb
        decision = 1
        break
    elseif p > ub
        decision = 0
        break
    end
end

return decision, p, t
end

abstract type HiddenDistribution end
struct F0 <: HiddenDistribution end
struct F1 <: HiddenDistribution end

"""
Uses the distribution f0 as the true data generating process
"""
function simulate_tdgp(wf::WaldFriedman, f::F0; p0::Real=0.5)
    decision, p, t = simulate(wf, wf.f0; p0=p0)
    correct = (decision == 0)
    return correct, p, t
end

"""
Uses the distribution f1 as the true data generating process
"""
function simulate_tdgp(wf::WaldFriedman, f::F1; p0::Real=0.5)
    decision, p, t = simulate(wf, wf.f1; p0=p0)
    correct = (decision == 1)
    return correct, p, t
end

"""
Simulates repeatedly to get distributions of time needed to make a
decision and how often they are correct.

```julia
function stopping_dist(wf::WaldFriedman;
                      ndraws::Integer=250, f::HiddenDistribution=F0())
    # Allocate space
    tdist = Vector{Int64}(ndraws)
    cdist = Vector{Bool}(ndraws)

    for i in 1:ndraws
        correct, p, t = simulate_tdgp(wf, f)
        tdist[i] = t
        cdist[i] = correct
    end

    return cdist, tdist
end
```

Now let's use our type to solve Bellman equation (3.18) and verify that it gives similar output

```julia
# Create two distributions over 50 values for k
# We are using a discretized beta distribution
p_m1 = linspace(0, 1, 50)
f0 = clamp.(pdf.(Beta(1, 1), p_m1), 1e-8, Inf)
f0 = f0 / sum(f0)
f1 = clamp.(pdf.(Beta(9, 9), p_m1), 1e-8, Inf)
f1 = f1 / sum(f1);
wf = WaldFriedman(0.5, 5.0, 5.0, f0, f1; m=251)
J2 = compute_fixed_point(x -> bellman_operator(wf, x), zeros(wf.m), err_ 
->tol=1e-6, print_skip=5)

@printf("If this is true then both approaches gave same answer:\n")
print(isapprox(J1, J2; atol=1e-5))
```

We get the same output in terms of distance

- Compute iterate 5 with error 0.0855260926408965
- Compute iterate 10 with error 0.00038782882545862485
- Compute iterate 15 with error 1.609783120581909e-6
- Converged in 16 steps

If this is true then both approaches gave same answer:
true

The approximate value functions produced are also the same

Rather than discuss this further, let's go ahead and use our code to generate some results

### 3.4.5 Analysis

Now that our routines are working, let's inspect the solutions

3.4. A Problem that Stumped Milton Friedman
Well start with the following parameterization

```julia
function analysis_plot(;c=1.25, L0=27.0, L1=27.0, a0=2.5, b0=2.0, a1=2.0,
  b1=2.5, m=251)
  f0 = pdf.(Beta(a0, b0), linspace(0, 1, m))
  f0 = f0 / sum(f0)
  f1 = pdf.(Beta(a1, b1), linspace(0, 1, m))
  f1 = f1 / sum(f1)  # Make sure sums to 1

  # Create an instance of our WaldFriedman class
  wf = WaldFriedman(c, L0, L1, f0, f1; m=m);

  # Solve and simulate the solution
  cdist, tdist = stopping_dist(wf; ndraws=5000)

  a = plot([f0 f1],
    xlabel=L"$s_k$ Values",
    ylabel=L"Probability of $s_k$",
    labels=[L"$s_0$" L"$s_1$"],
    linewidth=2,
    title="Distributions over Outcomes")

  b = plot(wf.pgrid, wf.sol.J,
    xlabel=L"$p_k$",
    ylabel="Value of Bellman",
    linewidth=2,
    title="Bellman Equation")
  plot!(fill(wf.sol.lb, 2), [minimum(wf.sol.J); maximum(wf.sol.J)],
    linewidth=2, color=:black, linestyle=:dash, label="", ann=(wf.sol.lb-
    0.05, 5., L"\beta")
  plot!(fill(wf.sol.ub, 2), [minimum(wf.sol.J); maximum(wf.sol.J)],
    linewidth=2, color=:black, linestyle=:dash, label="", ann=(wf.sol.
    ub+0.02, 5., L"\alpha"),
    legend=:none)

  counts = [sum(tdist .== i) for i in 1:maximum(tdist)]

  c = bar(counts,
    xticks=0:1:maximum(tdist),
    xlabel="Time",
    ylabel="Frequency",
    title="Stopping Times",
    legend=:none)

  counts = [sum(cdist .== i-1) for i in 1:2]

  d = bar([0; 1],
    counts,
    xticks=[0; 1],
    title="Correct Decisions",
    ann=(0.0, 0.6 * sum(cdist), "Percent Correct = $(sum(cdist)/
      length(cdist))")
    legend=:none)
```

QuantEcon.lectures-julia PDF, Release 2018-Aug-8
Heres a plot of some objects well discuss one by one

The code to generate this figure can be found in wald_solution_plots.jl

**Value Function**

In the top left subfigure we have the two beta distributions, \( f_0 \) and \( f_1 \)

In the top right we have corresponding value function \( J \)

It equals \( pL_1 \) for \( p \leq \beta \), and \( (1 - p)L_0 \) for \( p \geq \alpha \)

The slopes of the two linear pieces of the value function are determined by \( L_1 \) and \( -L_0 \)

The value function is smooth in the interior region, where the posterior probability assigned to \( f_0 \) is in the indecisive region \( p \in (\beta, \alpha) \)
The decision maker continues to sample until the probability that he attaches to model $f_0$ falls below $\beta$ or above $\alpha$

**Simulations**

The bottom two subfigures show the outcomes of 500 simulations of the decision process.

On the left is a histogram of the stopping times, which equal the number of draws of $z_k$ required to make a decision.

The average number of draws is around 6.6.

On the right is the fraction of correct decisions at the stopping time.

In this case the decision maker is correct 80% of the time.

**Comparative statics**

Now let's consider the following exercise.

We double the cost of drawing an additional observation.

Before you look, think about what will happen:

- Will the decision maker be correct more or less often?
- Will he make decisions sooner or later?

```
analysis_plot(c=2.5)
```

Here's the figure.
Notice what happens

The stopping times dropped dramatically!

Increased cost per draw has induced the decision maker usually to take only 1 or 2 draws before deciding

Because he decides with less, the percentage of time he is correct drops

This leads to him having a higher expected loss when he puts equal weight on both models

A notebook implementation

To facilitate comparative statics, we provide a Jupyter notebook that generates the same plots, but with sliders

With these sliders you can adjust parameters and immediately observe

- effects on the smoothness of the value function in the indecisive middle range as we increase the number of grid points in the piecewise linear approximation.

- effects of different settings for the cost parameters $L_0, L_1, c$, the parameters of two beta distributions $f_0$ and $f_1$, and the number of points and linear functions $m$ to use in the piece-wise continuous approximation to the value function.

3.4. A Problem that Stumped Milton Friedman


- various simulations from $f_0$ and associated distributions of waiting times to making a decision
- associated histograms of correct and incorrect decisions

### 3.4.6 Comparison with Neyman-Pearson formulation

For several reasons, it is useful to describe the theory underlying the test that Navy Captain G. S. Schuyler had been told to use and that led him to approach Milton Friedman and Allan Wallis to convey his conjecture that superior practical procedures existed.

Evidently, the Navy had told Captain Schuyler to use what it knew to be a state-of-the-art Neyman-Pearson test.


For our purposes, watch for there features of the setup:

- the assumption of a *fixed* sample size $n$
- the application of laws of large numbers, conditioned on alternative probability models, to interpret the probabilities $\alpha$ and $\beta$ defined in the Neyman-Pearson theory

Recall that in the sequential analytic formulation above, that

- The sample size $n$ is not fixed but rather an object to be chosen; technically $n$ is a random variable
- The parameters $\beta$ and $\alpha$ characterize cut-off rules used to determine $n$ as a random variable
- Laws of large numbers make no appearances in the sequential construction

In chapter 1 of *Sequential Analysis* [Wal47] Abraham Wald summarizes the Neyman-Pearson approach to hypothesis testing.

Wald frames the problem as making a decision about a probability distribution that is partially known.

(You have to assume that *something* is already known in order to state a well posed problem. Usually, *something* means a lot.)

By limiting what is unknown, Wald uses the following simple structure to illustrate the main ideas.

- a decision maker wants to decide which of two distributions $f_0, f_1$ govern an i.i.d. random variable $z$
- The null hypothesis $H_0$ is the statement that $f_0$ governs the data.
- The alternative hypothesis $H_1$ is the statement that $f_1$ governs the data.
- The problem is to devise and analyze a test of hypothesis $H_0$ against the alternative hypothesis $H_1$ on the basis of a sample of a fixed number $n$ independent observations $z_1, z_2, \ldots, z_n$ of the random variable $z$.

To quote Abraham Wald,

- A test procedure leading to the acceptance or rejection of the hypothesis in question is simply a rule specifying, for each possible sample of size $n$, whether the hypothesis should be accepted or rejected on the basis of the sample. This may also be expressed as follows: A test procedure is simply a subdivision of the totality of all possible samples of size $n$ into two mutually exclusive parts, say part 1 and part 2, together with the application of the rule that the hypothesis be accepted if the observed
sample is contained in part 2. Part 1 is also called the critical region. Since part 2 is the totality of all samples of size 2 which are not included in part 1, part 2 is uniquely determined by part 1. Thus, choosing a test procedure is equivalent to determining a critical region.

Lets listen to Wald longer:

- As a basis for choosing among critical regions the following considerations have been advanced by Neyman and Pearson: In accepting or rejecting \( H_0 \) we may commit errors of two kinds. We commit an error of the first kind if we reject \( H_0 \) when it is true; we commit an error of the second kind if we accept \( H_0 \) when \( H_1 \) is true. After a particular critical region \( W \) has been chosen, the probability of committing an error of the first kind, as well as the probability of committing an error of the second kind is uniquely determined. The probability of committing an error of the first kind is equal to the probability, determined by the assumption that \( H_0 \) is true, that the observed sample will be included in the critical region \( W \). The probability of committing an error of the second kind is equal to the probability, determined on the assumption that \( H_1 \) is true, that the probability will fall outside the critical region \( W \). For any given critical region \( W \) we shall denote the probability of an error of the first kind by \( \alpha \) and the probability of an error of the second kind by \( \beta \).

Let's listen carefully to how Wald applies a law of large numbers to interpret \( \alpha \) and \( \beta \):

- The probabilities \( \alpha \) and \( \beta \) have the following important practical interpretation: Suppose that we draw a large number of samples of size \( n \). Let \( M \) be the number of such samples drawn. Suppose that for each of these \( M \) samples we reject \( H_0 \) if the sample is included in \( W \) and accept \( H_0 \) if the sample lies outside \( W \). In this way we make \( M \) statements of rejection or acceptance. Some of these statements will in general be wrong. If \( H_0 \) is true and if \( M \) is large, the probability is nearly 1 (i.e., it is practically certain) that the proportion of wrong statements (i.e., the number of wrong statements divided by \( M \)) will be approximately \( \alpha \). If \( H_1 \) is true, the probability is nearly 1 that the proportion of wrong statements will be approximately \( \beta \). Thus, we can say that in the long run [here Wald applies a law of large numbers by driving \( M \to \infty \) (our comment, not Wald's)] the proportion of wrong statements will be \( \alpha \) if \( H_0 \) is true and \( \beta \) if \( H_1 \) is true.

The quantity \( \alpha \) is called the size of the critical region, and the quantity \( 1 - \beta \) is called the power of the critical region.

Wald notes that

- one critical region \( W \) is more desirable than another if it has smaller values of \( \alpha \) and \( \beta \). Although either \( \alpha \) or \( \beta \) can be made arbitrarily small by a proper choice of the critical region \( W \), it is possible to make both \( \alpha \) and \( \beta \) arbitrarily small for a fixed value of \( n \), i.e., a fixed sample size.

Wald summarizes Neyman and Pearson's setup as follows:

- Neyman and Pearson show that a region consisting of all samples \((z_1, z_2, \ldots, z_n)\) which satisfy the inequality

\[
\frac{f_1(z_1) \cdots f_1(z_n)}{f_0(z_1) \cdots f_1(z_n)} \geq k
\]

is a most powerful critical region for testing the hypothesis \( H_0 \) against the alternative hypothesis \( H_1 \). The term \( k \) on the right side is a constant chosen so that the region will have the required size \( \alpha \).
Wald goes on to discuss Neyman and Pearson’s concept of *uniformly most powerful* test.

Here is how Wald introduces the notion of a sequential test:

- A rule is given for making one of the following three decisions at any stage of the experiment (at the mth trial for each integral value of m): (1) to accept the hypothesis H, (2) to reject the hypothesis H, (3) to continue the experiment by making an additional observation. Thus, such a test procedure is carried out sequentially. On the basis of the first observation one of the aforementioned decisions is made. If the first or second decision is made, the process is terminated. If the third decision is made, a second trial is performed. Again, on the basis of the first two observations one of the three decisions is made. If the third decision is made, a third trial is performed, and so on. The process is continued until either the first or the second decisions is made. The number n of observations required by such a test procedure is a random variable, since the value of n depends on the outcome of the observations.

### 3.5 Job Search III: Search with Learning

**Contents**

- **Job Search III: Search with Learning**
  - Overview
  - Model
  - Take 1: Solution by VFI
  - Take 2: A More Efficient Method
  - Exercises
  - Solutions

#### 3.5.1 Overview

In this lecture we consider an extension of the *previously studied* job search model of McCall [McC70]. In the McCall model, an unemployed worker decides when to accept a permanent position at a specified wage, given

- his or her discount rate
- the level of unemployment compensation
- the distribution from which wage offers are drawn

In the version considered below, the wage distribution is unknown and must be learned

- The following is based on the presentation in [LS18], section 6.6
Model features

- Infinite horizon dynamic programming with two states and one binary control
- Bayesian updating to learn the unknown distribution

3.5.2 Model

Let's first review the basic McCall model [McC70] and then add the variation we want to consider.

The Basic McCall Model

Recall that, in the baseline model, an unemployed worker is presented in each period with a permanent job offer at wage $W_t$.

At time $t$, our worker either

1. accepts the offer and works permanently at constant wage $W_t$
2. rejects the offer, receives unemployment compensation $c$ and reconsiders next period

The wage sequence $\{W_t\}$ is iid and generated from known density $h$.

The worker aims to maximize the expected discounted sum of earnings $E \sum_{t=0}^{\infty} \beta^t y_t$. The function $V$ satisfies the recursion

$$V(w) = \max \left\{ \frac{w}{1 - \beta}, c + \beta \int V(w') h(w') dw' \right\} \quad (3.19)$$

The optimal policy has the form $1\{w \geq \bar{w}\}$, where $\bar{w}$ is a constant depending called the reservation wage.

Offer Distribution Unknown

Now let's extend the model by considering the variation presented in [LS18], section 6.6.

The model is as above, apart from the fact that

- the density $h$ is unknown
- the worker learns about $h$ by starting with a prior and updating based on wage offers that he/she observes

The worker knows there are two possible distributions $F$ and $G$ with densities $f$ and $g$.

At the start of time, nature selects $h$ to be either $f$ or $g$ the wage distribution from which the entire sequence $\{W_t\}$ will be drawn.

This choice is not observed by the worker, who puts prior probability $\pi_0$ on $f$ being chosen.

Update rule: workers time $t$ estimate of the distribution is $\pi_t f + (1 - \pi_t) g$, where $\pi_t$ updates via
\[ t_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t)g(w_{t+1})} \]  

This last expression follows from Bayes rule, which tells us that

\[ \mathbb{P}\{h = f | W = w\} = \frac{\mathbb{P}\{W = w | h = f\} \mathbb{P}\{h = f\}}{\mathbb{P}\{W = w\}} \quad \text{and} \quad \mathbb{P}\{W = w\} = \sum_{\psi \in \{f,g\}} \mathbb{P}\{W = w | h = \psi\} \mathbb{P}\{h = \psi\} \]

The fact that (3.20) is recursive allows us to progress to a recursive solution method. Letting

\[ h_\pi(w) := \pi f(w) + (1 - \pi)g(w) \quad \text{and} \quad q(w, \pi) := \frac{\pi f(w)}{\pi f(w) + (1 - \pi)g(w)} \]

we can express the value function for the unemployed worker recursively as follows

\[ V(w, \pi) = \max \left\{ \frac{w}{1 - \beta}, c + \beta \int V(w', \pi') h_\pi(w') \, dw' \right\} \quad \text{where} \quad \pi' = q(w', \pi) \]  

Notice that the current guess \( \pi \) is a state variable, since it affects the workers perception of probabilities for future rewards.

### Parameterization

Following section 6.6 of [LS18], our baseline parameterization will be

- \( f \) is Beta(1, 1) scaled (i.e., draws are multiplied by) some factor \( w_m \)
- \( g \) is Beta(3, 1.2) scaled (i.e., draws are multiplied by) the same factor \( w_m \)
- \( \beta = 0.95 \) and \( c = 0.6 \)

With \( w_m = 2 \), the densities \( f \) and \( g \) have the following shape:

```julia
using Distributions
using Plots

w_max = 2
x = linspace(0, w_max, 200)

G = Beta(3, 1.6)
F = Beta(1, 1)
plot!(x, pdf.(G, x/w_max)/w_max, label="g")
plot!(x, pdf.(F, x/w_max)/w_max, label="f")
```
Looking Forward

What kind of optimal policy might result from (3.21) and the parameterization specified above?

Intuitively, if we accept at $w_a$ and $w_a \leq w_b$, then all other things being given we should also accept at $w_b$. This suggests a policy of accepting whenever $w$ exceeds some threshold value $\bar{w}$.

But $\bar{w}$ should depend on $\pi$ in fact it should be decreasing in $\pi$ because

- $f$ is a less attractive offer distribution than $g$
- larger $\pi$ means more weight on $f$ and less on $g$

Thus larger $\pi$ depresses the workers assessment of her future prospects, and relatively low current offers become more attractive.

**Summary:** We conjecture that the optimal policy is of the form $\mathcal{K}\{w \geq \bar{w}(\pi)\}$ for some decreasing function $\bar{w}$. 

3.5. Job Search III: Search with Learning
3.5.3 Take 1: Solution by VFI

Let's set about solving the model and see how our results match with our intuition.

We begin by solving via value function iteration (VFI), which is natural but ultimately turns out to be second best.

The code is as follows:

```julia
#=
@author : Spencer Lyon <spencer.lyon@nyu.edu>
=#
using QuantEcon
using Interpolations

""
Unemployment/search problem where offer distribution is unknown

#### Fields
- `\beta`::Real` : Discount factor on (0, 1)
- `\beta`::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 offer offers
- `\quad_weights`::Vector` : Weights for quadrature offer offers

""
struct SearchProblem{TR<:Real, TI<:Integer, TF<:AbstractFloat,
  TAVw<:AbstractVector{TF}, TAVpi<:AbstractVector{TF}}
  \beta::TR
  c::TR
  F::Distribution
  G::Distribution
  f::Function
  g::Function
  \n_w::TI
  \w_max::TR
  \w_grid::TAVw
  \n_\pi::TI
  \pi_min::TR
  \pi_max::TR
```

Chapter 3. Dynamic Programming
π_grid::TAVpi
quad_nodes::Vector(TF)
quad_weights::Vector(TF)
end

""
Constructor for `SearchProblem` with default values

##### Arguments
- `β::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::Integer(40)` : Number of points in wage offer grid
- `π_grid_size::Integer(40)` : Number of points in probability grid

##### Notes
There is also a version of this function that accepts keyword arguments for each parameter

""

# use key word argument

function SearchProblem(β::Real=0.95, c::Real=0.6, F_a::Real=1, F_b::Real=1,
                      G_a::Real=3, G_b::Real=1.2, w_max::Real=2.0,
                      w_grid_size::Integer=40, π_grid_size::Integer=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

    π_min = 1e-3  # avoids instability
    π_max = 1 - π_min

    w_grid = linspace(0, w_max, w_grid_size)
    π_grid = linspace(π_min, π_max, π_grid_size)

    nodes, weights = qnwlege(21, 0.0, w_max)

    SearchProblem(β, c, F, G, f, g,
                   w_grid_size, w_max, w_grid,
                   π_grid_size, π_min, π_max, π_grid, nodes, weights)
end

function q(sp::SearchProblem, w, π_val)
    new_π = 1.0 / (1 + ((1 - π_val) * sp.g(w)) / (π_val * sp.f(w)))
end

3.5. Job Search III: Search with Learning
Apply the Bellman operator for a given model and initial value.

```julia
function bellman_operator!(sp::SearchProblem, v::Matrix, out::Matrix;
    ret_policy::Bool=false)
    # Simplify names
    f, g, β, c = sp.f, sp.g, sp.β, sp.c
    nodes, weights = sp.quad_nodes, sp.quad_weights

    vf = extrapolate(interpolate((sp.w_grid, sp.π_grid), v, Gridded(Linear())), Flat())

    # set up quadrature nodes/weights
    # q_nodes, q_weights = qnwlege(21, 0.0, sp.w_max)

    for (w_i, w) in enumerate(sp.w_grid)
        # calculate v1
        v1 = w / (1 - β)

        for (π_j, _π) in enumerate(sp.π_grid)
            # calculate v2
            integrand(m) = [vf[m[i], g.(sp, m[i], _π)] * (_π * f(m[i]) + (1 - _π) * g(m[i])) for i in 1:length(m)]
            integral = do_quad(integrand, nodes, weights)
            # integral = do_quad(integrand, q_nodes, q_weights)
            v2 = c + β * integral

            # return policy if asked for, otherwise return max of values
            out[w_i, π_j] = ret_policy ? v1 > v2 : max(v1, v2)
        end
    end
    return out
end
```

# Return new_π when in [π_min, π_max] and else end points
return clamp(new_π, sp.π_min, sp.π_max)
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_π)
    bellman_operator!(sp, v, out, ret_policy=ret_policy)
end

""
Extract the greedy policy (policy function) of the model.

##### 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

""
get_greedy!(sp::SearchProblem, v::Matrix, out::Matrix) =
    bellman_operator!(sp, v, out, ret_policy=true)
get_greedy(sp::SearchProblem, v::Matrix) =
    bellman_operator(sp, v, ret_policy=true)

""
Updates the reservation wage function guess via the operator Q.

##### Arguments
- `sp::SearchProblem`: Instance of `SearchProblem`
- `v::Vector`: Current guess for
- `out::Vector`: Storage for output

##### Returns
None, `out` is updated in place to hold the updated levels of

""
function res_wage_operator!(sp::SearchProblem, ::Vector, out::Vector)
    # Simplify name
    f, g, β, c = sp.f, sp.g, sp.β, sp.c

    # Construct interpolator over π_grid, given
    _f = LinInterp(sp.π_grid, )

    # set up quadrature nodes/weights
    q_nodes, q_weights = qnwege(7, 0.0, sp.w_max)

    for (i, _π) in enumerate(sp.π_grid)
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 at solutions computed from value function iteration.

Heres the value function:

```plaintext
using LaTeXStrings

# Set up the problem and initial guess, solve by VFI
sp = SearchProblem(;w_grid_size=100, \( \pi \_grid\_size=100 \))
v_init = zeros(sp.n_w, sp.n_\( \pi \)) + sp.c / (1 - sp.\( \beta \))
f(x) = bellman_operator(sp, x)
v = compute_fixed_point(f, v_init)
policy = get_greedy(sp, v)

# Make functions for the linear interpolants of these
vf = extrapolate(interpolate((sp.w_grid, sp.\( \pi \_grid \)), v, Gridded(Linear())), Flat())
pf = extrapolate(interpolate((sp.w_grid, sp.\( \pi \_grid \)), policy, Gridded(Linear())), Flat())

function plot_value_function(;w_plot_grid_size::Integer=100, \( \pi \_plot\_grid\_size::Integer=100 \))
    \( \pi \_plot\_grid \) = linspace(0.001, 0.99, \( \pi \_plot\_grid\_size \))
    w_plot_grid = linspace(0, sp.w_max, w_plot_grid_size)
    Z = [vf[w_plot_grid[i]], \( \pi \_plot\_grid[j] \)]
    for j in 1:w_plot_grid_size, i in 1:\( \pi \_plot\_grid\_size \)
```
The optimal policy:

```julia
function plot_policy_function(;w_plot_grid_size::Integer=100,
                                _plot_grid_size::Integer=100)

    π_plot_grid = linspace(0.001, 0.99, _plot_grid_size)
    w_plot_grid = linspace(0, sp.w_max, w_plot_grid_size)
    Z = [pf[w_plot_grid[j], π_plot_grid[i]] for j in 1:w_plot_grid_size, i in 1:_plot_grid_size]
    plot!(xlabel=L"\pi", ylabel="wage", xguidefont=font(12), cbar=false)
end
```

The optimal policy:
The code takes several minutes to run
The results fit well with our intuition from section *looking forward*

- The black line in the figure above corresponds to the function $\bar{w}(\pi)$ introduced there
- It is decreasing as expected

### 3.5.4 Take 2: A More Efficient Method

Our implementation of VFI can be optimized to some degree
But instead of pursuing that, let’s consider another method to solve for the optimal policy. We will use iteration with an operator that has the same contraction rate as the Bellman operator, but

- one dimensional rather than two dimensional
- no maximization step

As a consequence, the algorithm is orders of magnitude faster than VFI.

This section illustrates the point that when it comes to programming, a bit of mathematical analysis goes a long way.

### Another Functional Equation

To begin, note that when \( w = \bar{w}(\pi) \), the worker is indifferent between accepting and rejecting. Hence the two choices on the right-hand side of (3.21) have equal value:

\[
\frac{\bar{w}(\pi)}{1 - \beta} = c + \beta \int V(w', \pi') h(\pi) \, dw'
\]  

(3.22)

Together, (3.21) and (3.22) give

\[
V(w, \pi) = \max \left\{ \frac{w}{1 - \beta}, \frac{\bar{w}(\pi)}{1 - \beta} \right\}
\]

(3.23)

Combining (3.22) and (3.23), we obtain

\[
\frac{\bar{w}(\pi)}{1 - \beta} = c + \beta \int \max \left\{ \frac{w'}{1 - \beta}, \frac{\bar{w}(\pi')}{1 - \beta} \right\} h(\pi') \, dw'
\]

Multiplying by \( 1 - \beta \), substituting in \( \pi' = q(w', \pi) \) and using \( \circ \) for composition of functions yields

\[
\bar{w}(\pi) = (1 - \beta)c + \beta \int \max \left\{ \frac{w'}{1 - \beta}, \bar{w} \circ q(w', \pi) \right\} h(\pi) \, dw'
\]

(3.24)

Equation (3.24) 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

\[ (Q\psi)(\pi) = (1 - \beta)c + \beta \max \left\{ w', \psi \circ q(w', \pi) \right\} h_{\pi}(w') \, dw' \quad (3.25) \]

Comparing (3.24) and (3.25), we see that the set of fixed points of \( Q \) exactly coincides with the set of solutions to the RWFE

- If \( Q\bar{\psi} = \bar{\psi} \) then \( \bar{\psi} \) solves (3.24) and vice versa

Moreover, for any \( \psi, \phi \in b[0,1] \), basic algebra and the triangle inequality for integrals tells us that

\[ |(Q\psi)(\pi) - (Q\phi)(\pi)| \leq \beta \int |\max \left\{ w', \psi \circ q(w', \pi) \right\} - \max \left\{ w', \phi \circ q(w', \pi) \right\}| h_{\pi}(w') \, dw' \quad (3.26) \]

Working case by case, it is easy to check that for real numbers \( a, b, c \) we always have

\[ |\max\{a, b\} - \max\{a, c\}| \leq |b - c| \quad (3.27) \]

Combining (3.26) and (3.27) yields

\[ |(Q\psi)(\pi) - (Q\phi)(\pi)| \leq \beta \int |\psi \circ q(w', \pi) - \phi \circ q(w', \pi)| h_{\pi}(w') \, dw' \leq \beta \|\psi - \phi\| \quad (3.28) \]

Taking the supremum over \( \pi \) now gives us

\[ \|Q\psi - Q\phi\| \leq \beta \|\psi - \phi\| \quad (3.29) \]

In other words, \( Q \) is a contraction of modulus \( \beta \) on the complete metric space \((b[0,1], \| \cdot \|)\)

Hence

- A unique solution \( \bar{\psi} \) to the RWFE exists in \( b[0,1] \)
- \( Q^k\psi \to \bar{\psi} \) uniformly as \( k \to \infty \), for any \( \psi \in b[0,1] \)

**Implementation**

These ideas are implemented in the `.res_wage_operator()` method from `odu.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 \( \bar{\psi} \)
3.5.5 Exercises

Exercise 1

Use the default parameters and the .res_wage_operator() method to compute an optimal policy.

Your result should coincide closely with the figure for the optimal policy shown above.

Try experimenting with different parameters, and confirm that the change in the optimal policy coincides with your intuition.

3.5.6 Solutions

Exercise 1

This code solves the Offer Distribution Unknown model by iterating on a guess of the reservation wage function. You should find that the run time is much shorter than that of the value function approach in examples/odu_vfi_plots.jl.

```
sp = SearchProblem(\_grid_size=50)
\_init = ones(sp.n_\pi)
f_ex1(x) = res_wage_operator(sp, x)
w_bar = compute_fixed_point(f_ex1, \_init)

plot(sp.\_grid, w_bar, linewidth=2, color=:black, fill_between=0, fillalpha=0.15, fillcolor=:blue)
plot!(sp.\_grid, 2*ones(length(w_bar)), linewidth=0, fill_between=w_bar, fillalpha=0.12, fillcolor=:green, legend=:none)
plot!(ylims=(0, 2), annotations=[((0.42, 1.2), "reject"), (0.7, 1.8, "accept")])

Compute iterate 10 with error 0.007194437603255555
Compute iterate 20 with error 0.0004348703417873523
Converged in 26 steps
```
The next piece of code is not one of the exercises from QuantEcon – its just a fun simulation to see what the effect of a change in the underlying distribution on the unemployment rate is

At a point in the simulation, the distribution becomes significantly worse. It takes a while for agents to learn this, and in the meantime they are too optimistic, and turn down too many jobs. As a result, the unemployment rate spikes

The code takes a few minutes to run.

```julia
# Set up model and compute the function w_bar
sp = SearchProblem(π_grid_size=50, F_a=1, F_b=1)
_init = ones(sp.n_π)
g(x) = res_wage_operator(sp, x)
w_bar_vals = compute_fixed_point(g, _init)
w_bar = extrapolate(interpolate((sp.π_grid, ), w_bar_vals, Gridded(Linear())), Flat())

# Holds the employment state and beliefs of an individual agent.
mutable struct Agent
    _π
    employed
end
Agent(_π=1e-3) = Agent(_π, 1)

function update!(ag::Agent, H::Distribution)
    if ag.employed == 0
        w = rand(H) * 2  # account for scale in julia
    end
```
if $w \geq w_{\text{bar}}[ag.\_\pi]$ 
    ag.employed = 1
else
    ag.\_\pi = 1.0 ./ (1 + ((1 - ag.\_\pi) \ast sp.g(w)) ./ (ag.\_\pi + sp.\_f(w)))
end
end
nothing
end
num_agents = 5000
separation_rate = 0.025  # Fraction of jobs that end in each period
separation_num = round(Int, num_agents * separation_rate)
agent_indices = collect(1:num_agents)
agents = [Agent() for i=1:num_agents]
sim_length = 600
H = sp.G  # Start with distribution G
change_date = 200  # Change to F after this many periods
unempl_rate = Vector{Float64}(sim_length)
for i=1:sim_length
    if i % 20 == 0
        println("date = "$i"
    end
    if i == change_date
        H = sp.F
    end
    # Randomly select separation_num agents and set employment status to 0
    shuffle!(agent_indices)
    separation_list = agent_indices[1:separation_num]
    for agent in agents[separation_list]
        agent.employed = 0
    end
    # update agents
    for agent in agents
        update!(agent, H)
    end
    employed = Int[agent.employed for agent in agents]
    unempl_rate[i] = 1.0 - mean(employed)
end
plot(unempl_rate, linewidth=2, label="unemployment rate")
vline!([change_date], color=:red, label="")

Compute iterate 10 with error 0.00719437603255555
Compute iterate 20 with error 0.0004348703417873523
Converged in 26 steps
date = 20
date = 40

3.5. Job Search III: Search with Learning
3.6 Job Search IV: Modeling Career Choice

Contents

• Job Search IV: Modeling Career Choice
  – Overview
  – Model
  – Implementation: career.jl
  – Exercises
  – Solutions

3.6.1 Overview

Next we study a computational problem concerning career and job choices.

The model is originally due to Derek Neal [Nea99]

This exposition draws on the presentation in [LS18], section 6.5

Model features

• Career and job within career both chosen to maximize expected discounted wage flow
• Infinite horizon dynamic programming with two state variables

3.6.2 Model

In what follows we distinguish between a career and a job, where

• a career is understood to be a general field encompassing many possible jobs, and
• a job is understood to be a position with a particular firm

For workers, wages can be decomposed into the contribution of job and career

• \( w_t = \theta_t + \epsilon_t \), where
  – \( \theta_t \) is contribution of career at time \( t \)
  – \( \epsilon_t \) is contribution of job at time \( t \)

At the start of time \( t \), a worker has the following options

• retain a current (career, job) pair \( (\theta_t, \epsilon_t) \) referred to hereafter as stay put
• retain a current career \( \theta_t \) but redraw a job \( \epsilon_t \) referred to hereafter as new job
• redraw both a career \( \theta_t \) and a job \( \epsilon_t \) referred to hereafter as new life
Draws of $\theta$ and $\epsilon$ are independent of each other and past values, with

- $\theta_t \sim F$
- $\epsilon_t \sim G$

Notice that the worker does not have the option to retain a job but redraw a career, starting a new career always requires starting a new job.

A young worker aims to maximize the expected sum of discounted wages

$$E \sum_{t=0}^{\infty} \beta^t w_t$$  \hfill (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

$$I = \theta + \epsilon + \beta V(\theta, \epsilon)$$
$$II = \theta + \int \epsilon' G(d\epsilon') + \beta \int V(\theta, \epsilon') G(d\epsilon')$$
$$III = \int \theta' F(d\theta') + \int \epsilon' G(d\epsilon') + \beta \int \int V(\theta', \epsilon') G(d\epsilon') F(d\theta')$$

Evidently $I$, $II$ and $III$ correspond to stay put, new job and new life, respectively.

**Parameterization**

As in [LS18], section 6.5, we will focus on a discrete version of the model, parameterized as follows:

- both $\theta$ and $\epsilon$ take values in the set $\text{linspace}(0, B, N)$ an even grid of $N$ points between 0 and $B$ inclusive
- $N = 50$
- $B = 5$
- $\beta = 0.95$

The distributions $F$ and $G$ are discrete distributions generating draws from the grid points $\text{linspace}(0, B, N)$.

A very useful family of discrete distributions is the Beta-binomial family, with probability mass function

$$p(k | n, a, b) = \binom{n}{k} \frac{B(k + a, n - k + b)}{B(a, b)}, \quad k = 0, \ldots, n$$
Interpretation:

- draw $q$ from a $\beta$ distribution with shape parameters $(a, b)$
- run $n$ independent binary trials, each with success probability $q$
- $p(k | n, a, b)$ is the probability of $k$ successes in these $n$ trials

Nice properties:

- very flexible class of distributions, including uniform, symmetric unimodal, etc.
- only three parameters

Here’s a figure showing the effect of different shape parameters when $n = 50$

```julia
using PyPlot
using QuantEcon
using Distributions

n = 50
a_vals = [0.5, 1, 100]
b_vals = [0.5, 1, 100]

fig, ax = plt.subplots(figsize=(8, 5))
for (a, b) in zip(a_vals, b_vals)
    ab_label = latexstring("a = $a$, b = $b$")
    dist = BetaBinomial(n, a, b)
    ax.plot(0:n, pdf(dist, support(dist)), "o", label=ab_label)
end
ax.legend()
```

![Plot showing the effect of different shape parameters](image-url)
3.6.3 Implementation: career.jl

The code for solving the DP problem described above is found in this file, which is repeated here for convenience.

```julia
""
Career/job choice model of Derek Neal (1999)

### Fields

- `β::AbstractFloat` : Discount factor in (0, 1)
- `N::Integer` : Number of possible realizations of both and θ
- `B::AbstractFloat` : upper bound for both and θ
- `θ::AbstractVector` : A grid of values on [0, B]
- `::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::AbstractFloat` : The mean of the distribution F
- `G_mean::AbstractFloat` : The mean of the distribution G

""

struct CareerWorkerProblem( TF <: AbstractFloat, 
   TI <: Integer, 
   TAV <: AbstractVector{TF}, 
   TAV2 <: AbstractVector{TF})

   β::TF
   N::TI
   B::TF
   θ::TAV
   ::TAV
   F_probs::TAV2
   G_probs::TAV2
   F_mean::TF
   G_mean::TF

end

""

Constructor with default values for `CareerWorkerProblem`

### Arguments

- `β::Real(0.95)` : Discount factor in (0, 1)
- `B::Real(5.0)` : upper bound for both and θ
- `N::Real(50)` : Number of possible realizations of both and θ
- `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

There is also a version of this function that accepts keyword arguments for each parameter.
```
function CareerWorkerProblem{T<:AbstractFloat}(β::Float64 = 0.95, 
B::Float64 = 5.0, 
N::Integer = 50, 
F_a::Float64 = 1.0, 
F_b::Float64 = 1.0, 
G_a::Float64 = 1.0, 
G_b::Float64 = 1.0)

θ = linspace(0, B, N) = copy(θ)

distinct_F = BetaBinomial(N-1, F_a, F_b)
dist_G = BetaBinomial(N-1, G_a, G_b)
F_probs = pdf(dist_F, support(dist_F))
G_probs = pdf(dist_G, support(dist_G))
F_mean = sum(θ .* F_probs)
G_mean = sum(θ .* G_probs)

CareerWorkerProblem(β, N, B, θ, F_probs, G_probs, F_mean, G_mean)

end

Apply the Bellman operator for a given model and initial value.

### 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 update_bellman!(cp::CareerWorkerProblem, 

    v::Array,
    out::Array;
    ret_policy::Bool=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.β) * 
    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
            vi = cp.θ[i] + cp.[j] + cp.β * v[i, j]
        end
    end

end
# new job
v2 = (cp.θ[i] + cp.G_mean + cp.β .* 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
    out[i, j] = action
else
    out[i, j] = max(v1, v2, v3)
end
end

def update_bellman(cp::CareerWorkerProblem, v::Array; ret_policy::Bool=false)
    out = similar(v)
    update_bellman!(cp, v, out, ret_policy=ret_policy)
    return out
end

""
Extract the greedy policy (policy function) of the model.

#### 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

""
def get_greedy!(cp::CareerWorkerProblem, v::Array, out::Array)
    update_bellman!(cp, v, out, ret_policy=true)
end

def get_greedy(cp::CareerWorkerProblem, v::Array)
    update_bellman(cp, v, ret_policy=true)
end

The code defines
• a type `CareerWorkerProblem` that
  – encapsulates all the details of a particular parameterization
  – implements the Bellman operator $T$

  In this model, $T$ is defined by $Tv(\theta, \epsilon) = \max\{I, II, III\}$, where $I$, $II$ and $III$ are as given in (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 [LS18], section 6.5.

  Hence we can reproduce figures 6.5.1 and 6.5.2 shown there, which exhibit the value function and optimal policy respectively

  Here's the value function

```julia
wp = CareerWorkerProblem()
v_init = fill(100.0, wp.N, wp.N)
func(x) = update_bellman(wp, x)
v = compute_fixed_point(func, v_init, max_iter=500, verbose=false)

# === plot value function === #
tg, eg = meshgrid(wp.θ, wp.)
surf(tg,
    eg,
    v',
    rstride=2,
    cstride=2,
    cmap="jet",
    alpha=0.5,
    linewidth=0.25)

ax = plt[:gca]()
ax[:set_zlim](150, 200)
ax[:set_xlabel]("θ")
ax[:set_ylabel]("")
ax[:view_init](ax[:elev], 225)
```
Fig. 3.1: Value function with uniform probabilities

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
3.6.4 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

Lets now consider how long it takes for the worker to settle down to a permanent job, given a starting point of $(\theta, \epsilon) = (0, 0)$.

In other words, we want to study the distribution of the random variable

$$T^* := \text{the first point in time from which the worker’s job no longer changes}$$
Evidently, the workers job becomes permanent if and only if $(\theta_t, \epsilon_t)$ enters the stay put region of $(\theta, \epsilon)$ space. Letting $S$ denote this region, $T^*$ can be expressed as the first passage time to $S$ under the optimal policy:

$$T^* := \inf \{ t \geq 0 \mid (\theta_t, \epsilon_t) \in S \}$$

Collect 25,000 draws of this random variable and compute the median (which should be about 7).

Repeat the exercise with $\beta = 0.99$ and interpret the change.

**Exercise 3**

As best you can, reproduce the figure showing the optimal policy.

Hint: The `get_greedy()` method returns a representation of the optimal policy where values 1, 2 and 3 correspond to stay put, new job and new life respectively. Use this and `contourf` from PyPlot.jl to produce the different shadings.

Now set $G_a = G_b = 100$ and generate a new figure with these parameters. Interpret.

### 3.6.5 Solutions

**Exercise 1**

```julia
wp = CareerWorkerProblem()

function solve_wp(wp::CareerWorkerProblem)
    v_init = fill(100.0, wp.N, wp.N)
    func(x) = update_bellman(wp, x)
    v = compute_fixed_point(func, v_init, max_iter=500, verbose=false)
    optimal_policy = get_greedy(wp, v)
    return v, optimal_policy
end

v, optimal_policy = solve_wp(wp)

F = DiscreteRV(wp.F_probs)
G = DiscreteRV(wp.G_probs)

function gen_path(T=20)
    i = j = 1
    θ_ind = Int[]
    _ind = Int[]

    for t=1:T
        # do nothing if stay put
        if optimal_policy[i, j] == 2 # new job
            j = rand(G)[1]
        elseif optimal_policy[i, j] == 3 # new life
            i, j = rand(F)[1], rand(G)[1]
        end
    end
```

3.6. Job Search IV: Modeling Career Choice
Exercise 2

The median for the original parameterization can be computed as follows

```julia
function gen_first_passage_time(optimal_policy::Matrix)
    t = 0
    i = j = 1
    while true
        push!(θ_ind, i)
        push!(θ_ind, j)
    end
    return wp.θ[θ_ind], wp.[θ_ind]
end
```

```julia
fig, axes = plt[:subplots](2, 1, figsize=(10, 8))
for ax in axes
    θ_path, _path = gen_path()
    ax[:plot](_path, label="")
    ax[:plot](θ_path, label="θ")
    ax[:legend](loc="lower right")
end
```
if optimal_policy[i, j] == 1  # Stay put
    return t
elseif optimal_policy[i, j] == 2  # New job
    j = rand(G)[1]
else  # New life
    i, j = rand(F)[1], rand(G)[1]
end
t += 1
end

M = 25000
samples = Array{Float64}(M)
for i=1:M
    samples[i] = gen_first_passage_time(optimal_policy)
end
print(median(samples))

7.0

To compute the median with $\beta = 0.99$ instead of the default value $\beta = 0.95$, replace
wp=CareerWorkerProblem() with wp=CareerWorkerProblem(\beta=0.99)
The medians are subject to randomness, but should be about 7 and 11 respectively. Not surprisingly, more
patient workers will wait longer to settle down to their final job

wp2 = CareerWorkerProblem(\beta=0.99)
v2, optimal_policy2 = solve_wp(wp2)
samples2 = Array{Float64}(M)
for i=1:M
    samples2[i] = gen_first_passage_time(optimal_policy2)
end
print(median(samples2))

14.0

Exercise 3

Here's the code to reproduce the original figure

fig, ax = plt[:,subplots](figsize=(6, 6))

lvls = [0.5, 1.5, 2.5, 3.5]
ax[:,contourf](tg, eg, optimal_policy', levels=lvls, cmap="winter", alpha=0.5)
ax[:,contour](tg, eg, optimal_policy', colors="k", levels=lvls, linewidths=2)
ax[:,set_xlabel]("\theta", fontsize=14)
ax[:,set_ylabel]("", fontsize=14)
Now we want to set \( G_a = G_b = 100 \) and generate a new figure with these parameters.

To do this, replace:

\[
wp = \text{CareerWorkerProblem()}
\]

with:

\[
wp = \text{CareerWorkerProblem}(G_a=100.0, G_b=100.0)
\]

In the new figure, you will see that the region for which the worker will stay put has grown because the distribution for \( \epsilon \) has become more concentrated around the mean, making high-paying jobs less realistic.

### 3.7 Job Search V: On-the-Job Search

Contents

- Job Search V: On-the-Job Search
  - Overview
3.7.1 Overview

In this section we solve a simple on-the-job search model

- based on [LS18], exercise 6.18, and [Jov79]

Model features

- job-specific human capital accumulation combined with on-the-job search
- infinite horizon dynamic programming with one state variable and two controls

3.7.2 Model

Let

- \( x_t \) denote the time-\( t \) job-specific human capital of a worker employed at a given firm
- \( w_t \) denote current wages

Let \( w_t = x_t(1 - s_t - \phi_t) \), where

- \( \phi_t \) is investment in job-specific human capital for the current role
- \( s_t \) is search effort, devoted to obtaining new offers from other firms.

For as long as the worker remains in the current job, evolution of \( \{x_t\} \) is given by \( x_{t+1} = G(x_t, \phi_t) \)

When search effort at \( t \) is \( s_t \), the worker receives a new job offer with probability \( \pi(s_t) \in [0, 1] \)

Value of offer is \( U_{t+1} \), where \( \{U_t\} \) is iid with common distribution \( F \)

Worker has the right to reject the current offer and continue with existing job

In particular, \( x_{t+1} = U_{t+1} \) if accepts and \( x_{t+1} = G(x_t, \phi_t) \) if rejects

Letting \( b_{t+1} \in \{0, 1\} \) be binary with \( b_{t+1} = 1 \) indicating an offer, we can write

\[
x_{t+1} = (1 - b_{t+1})G(x_t, \phi_t) + b_{t+1} \max\{G(x_t, \phi_t), U_{t+1}\}
\]  \hspace{1cm} (3.31)

Agents objective: maximize expected discounted sum of wages via controls \( \{s_t\} \) and \( \{\phi_t\} \)
Taking the expectation of $V(x_{t+1})$ and using (3.31), the Bellman equation for this problem can be written as

$$V(x) = \max_{s+\phi \leq 1} \left\{ x(1-s-\phi) + \beta(1-\pi(s))V[G(x,\phi)] + \beta \pi(s) \int V[G(x,\phi) \vee u]F(du) \right\}.$$  \hspace{1cm} (3.32)

Here nonnegativity of $s$ and $\phi$ is understood, while $a \vee b := \max\{a,b\}$

**Parameterization**

In the implementation below, we will focus on the parameterization

$$G(x,\phi) = A(x\phi)^\alpha, \quad \pi(s) = \sqrt{s} \quad \text{and} \quad F = \text{Beta}(2,2)$$

with default parameter values

- $A = 1.4$
- $\alpha = 0.6$
- $\beta = 0.96$

The Beta(2,2) distribution is supported on $(0, 1)$. It has a unimodal, symmetric density peaked at 0.5

**Back-of-the-Envelope Calculations**

Before we solve the model, let's make some quick calculations that provide intuition on what the solution should look like.

To begin, observe that the worker has two instruments to build capital and hence wages:

1. invest in capital specific to the current job via $\phi$
2. search for a new job with better job-specific capital match via $s$

Since wages are $x(1-s-\phi)$, marginal cost of investment via either $\phi$ or $s$ is identical.

Our risk neutral worker should focus on whatever instrument has the highest expected return.

The relative expected return will depend on $x$.

For example, suppose first that $x = 0.05$

- If $s = 1$ and $\phi = 0$, then since $G(x,\phi) = 0$, taking expectations of (3.31) gives expected next period capital equal to $\pi(s)\mathbb{E}U = \mathbb{E}U = 0.5$
- If $s = 0$ and $\phi = 1$, then next period capital is $G(x,\phi) = G(0.05, 1) \approx 0.23$

Both rates of return are good, but the return from search is better.

Next suppose that $x = 0.4$

- If $s = 1$ and $\phi = 0$, then expected next period capital is again $0.5$
- If $s = 0$ and $\phi = 1$, then $G(x,\phi) = G(0.4, 1) \approx 0.8$
Return from investment via $\phi$ dominates expected return from search

Combining these observations gives us two informal predictions:

1. At any given state $x$, the two controls $\phi$ and $s$ will function primarily as substitutes worker will focus on whichever instrument has the higher expected return

2. For sufficiently small $x$, search will be preferable to investment in job-specific human capital. For larger $x$, the reverse will be true

Now lets turn to implementation, and see if we can match our predictions

### 3.7.3 Implementation

The following code solves the DP problem described above

```julia
#=
@author: Spencer Lyon <spencer.lyon@nyu.edu>
=#
using Distributions
using QuantEcon

# NOTE: only brute-force approach is available in bellman operator.
# Waiting on a simple constrained optimizer to be written in pure Julia

""
A Jovanovic-type model of employment with on-the-job search.
The value function is given by
\[ V(x) = \max_\{s\} w(x, s) \]

for
\[ w(x, s) := x(1 - s) + \beta (1 - \pi(s)) V(G(x, s)) + \beta \pi(s) E \max(G(x, s), U) \]

where

* `x`: human capital
* `s`: search effort
* `\pi(s)`: investment in human capital
* `\pi(s)`: probability of new offer given search level $s$
* `x(1 - s)`: wage
* `G(x, s)`: new human capital when current job retained
* `U`: Random variable with distribution $F$ -- new draw of human capital

#### Fields

- `A::Real`: Parameter in human capital transition function
```
- `α::Real` : Parameter in human capital transition function
- `β::AbstractFloat` : Discount factor in (0, 1)
- `x_grid::AbstractVector` : Grid for potential levels of x
- `G::Function` : Transition `function` for human capital
- `π_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
- `quad_weights::Vector` : Quadrature weights for integrating over
- `::AbstractFloat` : A small number, used in optimization routine

```
struct JvWorker(TR <: Real,
               TF <: AbstractFloat,
               TUD <: UnivariateDistribution,
               TAV <: AbstractVector,
               TV <: Vector)
  A::TR
  α::TR
  β::TF
  x_grid::TAV
  G::Function
  π_func::Function
  F::TUD
  quad_nodes::TV
  quad_weights::TV::TF
end
```

Constructor with default values for `JvWorker`

```
#### Arguments

- `A::Real(1.4)` : Parameter in human capital transition function
- `α::Real(0.6)` : Parameter in human capital transition function
- `β::Real(0.96)` : Discount factor in (0, 1)
- `grid_size::Integer(50)` : Number of points in discrete grid for `x`
- `::Float(1e-4)` : A small number, used in optimization routine

#### Notes

There is also a version of this function that accepts keyword arguments for
each parameter

```
# use key word argument
function JvWorker(;A::Real=1.4,
                   α::Real=0.6,
                   β::Real=0.96,
                   grid_size::Integer=50,
                   ...)
::AbstractFloat=1e-4)

G(x, ) = A .* (x .*).^α
π_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 - α))) / (1.0 - α)), quantile(F, 1 - )

# range for linspace(, grid_max, grid_size). Needed for
# CoordInterpGrid below
x_grid = linspace(x, grid_max, grid_size)

JvWorker(A, α, β, x_grid, G, π_func, F, nodes, weights, )

""
Apply the Bellman operator for a given model and initial value,
returning only the value function

##### Arguments
- `jv::JvWorker` : Instance of `JvWorker`
- `V::Vector` : Current guess for the value function
- `new_V::Vector` : Storage for updated value function

##### Returns
None, `new_V` is updated in place with the value function.

##### Notes
Currently, only the brute-force approach is available.
We are waiting on a simple constrained optimizer to be written in pure Julia

""
function bellman_operator!(jv::JvWorker, V::AbstractVector, new_→V::AbstractVector)

  # simplify notation
  nodes, weights = jv.quad_nodes, jv.quad_weights

  # prepare interpoland of value function

3.7. Job Search V: On-the-Job Search
Vf = LinInterp(jv.x_grid, V)

# instantiate the linesearch variables
max_val = -1.0
cur_val = 0.0
max_s = 1.0
max_ = 1.0
search_grid = linspace(, 1.0, 15)

for (i, x) in enumerate(jv.x_grid)

    function w(z)
        s, = z
        h(u) = [Vf(max(G(x, ), uval)) * pdf(F, uval) for uval in u]  
        integral = do_quad(h, nodes, weights)
        q = π_func(s) * integral + (1.0 - π_func(s)) * Vf(G(x, ))
        return - x * (1.0 - - s) − β * q
    end

    for s in search_grid
        for in search_grid
            cur_val = elseif(s + <= 1.0, -w((s, )), -1.0)
            if cur_val > max_val
                max_val, max_s, max_ = cur_val, s,
            end
        end
    end

    new_V[i] = max_val
end

""
Apply the Bellman operator for a given model and initial value, returning

policies

### Arguments

- `jv::JvWorker` : Instance of `JvWorker`
- `V::Vector`: Current guess for the value function
- `out::Tuple(Vector, Vector)`: Storage for the two policy rules

### Returns

None, `out` is updated in place with the two policy functions.

### Notes

Currently, only the brute-force approach is available.
We are waiting on a simple constrained optimizer to be written in pure Julia

""
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

$$w(s, \phi) := -\left\{x(1 - s - \phi) + \beta(1 - \pi(s))V[G(x, \phi)] + \beta\pi(s) \int V[G(x, \phi) \lor u]F(du)\right\}$$

(3.33)

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 $x_{\text{grid}}$.

But to evaluate the right-hand side of (3.33), we need a function, so we replace the arrays $V$ and $x_{\text{grid}}$ with a function $Vf$ that gives linear interpolation of $V$ on $x_{\text{grid}}$.

Hence in the preliminaries of `bellman_operator`
• from the array $V$ we define a linear interpolation $Vf$ of its values
  – $c1$ is used to implement the constraint $s + \phi \leq 1$
  – $c2$ is used to implement $s \geq \epsilon$, a numerically stable alternative to the true constraint $s \geq 0$
  – $c3$ does the same for $\phi$

Inside the `for` loop, for each $x$ in the grid over the state space, we set up the function $w(z) = w(s, \phi)$ defined in (3.33).

The function is minimized over all feasible $(s, \phi)$ pairs, either by
• a relatively sophisticated solver from SciPy called `fmin_slsqp`, or
• brute force search over a grid

The former is much faster, but convergence to the global optimum is not guaranteed. Grid search is a simple way to check results.
### 3.7.4 Solving for Policies

Let's plot the optimal policies and see what they look like.

The code is as follows:

```julia
using LaTeXStrings
using Plots
using Plots.PlotMeasures
pyplot()

wp = JvWorker(grid_size=25)
_vinit = wp.x_grid .* 0.5

f(x) = bellman_operator(wp, x)
V = compute_fixed_point(f, v_init, max_iter=300)

_s_policy, _policy = bellman_operator(wp, V, ret_policies=true)

# === plot solution === #
p = plot(wp.x_grid, [_policy s_policy V],
     title=[" policy" "s policy" "value function"],
     color=[:orange :blue :green],
     xaxis="x", (0.0, maximum(wp.x_grid))),
     yaxis=((-.1, 1.1)), size=(800, 800),
     legend=false, layout=(3, 1),
     bottom_margin=Measures.Length(:mm, 20))
```

It produces the following figure.
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
3.7.5 Exercises

Exercise 1

Let's look at the dynamics for the state process \( \{x_t\} \) associated with these policies. The dynamics are given by (3.31) 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 \( \text{plot\_grid} \), 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

\[
K = 50 \\
\text{plot\_grid\_max, plot\_grid\_size} = 1.2, 100 \\
\text{plot\_grid} = \text{linspace}(0, \text{plot\_grid\_max}, \text{plot\_grid\_size}) \\
\text{plot} \left( \text{plot\_grid}, \text{plot\_grid}, \text{color}=:black, \text{linestyle}=:dash, \right. \\
\left. \text{lims}=(0, \text{plot\_grid\_max}), \text{legend}=:none \right)
\]

By examining the plot, argue that under the optimal policies, the state \( x_t \) will converge to a constant value \( \bar{x} \) close to unity.

Argue that at the steady state, \( s_t \approx 0 \) and \( \phi_t \approx 0.6 \).

Exercise 2

In the preceding exercise we found that \( s_t \) converges to zero and \( \phi_t \) converges to about 0.6.

Since these results were calculated at a value of \( \beta \) close to one, let's compare them to the best choice for an infinitely patient worker.

Intuitively, an infinitely patient worker would like to maximize steady state wages, which are a function of steady state capital.

You can take it as given that the infinitely patient worker does not search in the long run (i.e., \( s_t = 0 \) for large \( t \)).

Thus, given \( \phi \), steady state capital is the positive fixed point \( x^*(\phi) \) of the map \( x \mapsto G(x, \phi) \).

Steady state wages can be written as \( w^*(\phi) = x^*(\phi)(1 - \phi) \).

Graph \( w^*(\phi) \) with respect to \( \phi \), and examine the best choice of \( \phi \).

Can you give a rough interpretation for the value that you see?

3.7.6 Solutions

Exercise 1

Here's code to produce the 45 degree diagram.
wp = JvWorker(grid_size=25)
G, π_func, F = wp.G, wp.π_func, wp.F  # Simplify names

v_init = collect(wp.x_grid) * 0.5
println("Computing value function")
f2(x) = bellman_operator(wp, x)
V = compute_fixed_point(f2, v_init, max_iter=300)
println("Computing policy functions")
s_policy, _policy = bellman_operator(wp, V, ret_policies=true)

# Turn the policy function arrays into CoordInterpGrid objects for interpolation
s = LinInterp(wp.x_grid, s_policy)
   = LinInterp(wp.x_grid, _policy)

h_func(x, b, U) = (1 - b) * G(x, (x)) + b * max(G(x, (x)), U)

Computing value function
Compute iterate 10 with error 0.27832355616098514
Compute iterate 20 with error 0.18485362760592938
Compute iterate 30 with error 0.1227388307811066
Compute iterate 40 with error 0.08154249695446047
Compute iterate 50 with error 0.05415792547131559
Compute iterate 60 with error 0.0359696659293694
Compute iterate 70 with error 0.023890104457239048
Compute iterate 80 with error 0.015867045344705843
Compute iterate 90 with error 0.01053835398092487
Compute iterate 100 with error 0.006999259432742377
Compute iterate 110 with error 0.004648684856004337
Compute iterate 120 with error 0.003087508199701716
Compute iterate 130 with error 0.00205062462975345
Compute iterate 140 with error 0.0013619593579647926
Compute iterate 150 with error 0.000904569962393964
Compute iterate 160 with error 0.0006007865154593617
Compute iterate 170 with error 0.0003990232399502247
Compute iterate 180 with error 0.0002650185081300187
Compute iterate 190 with error 0.00017601683965118298
Compute iterate 200 with error 0.00011690477038861502
Converged in 204 steps
Computing policy functions

K = 50

plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = linspace(0, plot_grid_max, plot_grid_size)
ticks = [0.25, 0.5, 0.75, 1.0]
xs = []
ys = []
for x in plot_grid
    for i=1:K
        b = rand() < π_func(s(x)) ? 1 : 0
Looking at the dynamics, we can see that

- If $x_t$ is below about 0.2 the dynamics are random, but $x_{t+1} > x_t$ is very likely
- As $x_t$ increases the dynamics become deterministic, and $x_t$ converges to a steady state value close to $1$

Referring back to the figure here

http://quant-econ.net/jl/jv.html#solving-for-policies

we see that $x_t \approx 1$ means that $s_t = s(x_t) \approx 0$ and $\phi_t = \phi(x_t) \approx 0.6$
Exercise 2

```julia
wp = JvWorker(grid_size=25)
xbar() = (wp.A * wp.α)^(1.0 / (1.0 - wp.α))
_grid = linspace(0, 1, 100)
plot(_grid, [xbar() * (1 - ) for in _grid], color=blue,
     label=L"$w^* (\phi)$", legendfont=font(12), xlabel=L"$\phi$",
     guidefont=font(16), grid=false, legend=:topleft)
```

Observe that the maximizer is around 0.6
This this is similar to the long run value for $\phi$ obtained in exercise 1
Hence the behaviour of the infinitely patent worker is similar to that of the worker with $\beta = 0.96$
This seems reasonable, and helps us confirm that our dynamic programming solutions are probably correct

3.8 Optimal Growth I: The Stochastic Optimal Growth Model
3.8.1 Overview

In this lecture we are going to study a simple optimal growth model with one agent. The model is a version of the standard one sector infinite horizon growth model studied in

- [SLP89], chapter 2
- [LS18], section 3.1
- EDTC, chapter 1
- [Sun96], chapter 12

The technique we use to solve the model is dynamic programming. Our treatment of dynamic programming follows on from earlier treatments in our lectures on shortest paths and job search.

We will discuss some of the technical details of dynamic programming as we go along.

3.8.2 The Model

Consider an agent who owns an amount \( y_t \in \mathbb{R}_+ := [0, \infty) \) of a consumption good at time \( t \).

This output can either be consumed or invested.

When the good is invested it is transformed one-for-one into capital.

The resulting capital stock, denoted here by \( k_{t+1} \), will then be used for production.

Production is stochastic, in that it also depends on a shock \( \xi_{t+1} \) realized at the end of the current period.

Next period output is

\[
y_{t+1} := f(k_{t+1})\xi_{t+1}
\]

where \( f: \mathbb{R}_+ \rightarrow \mathbb{R}_+ \) is called the production function.

The resource constraint is

\[
k_{t+1} + c_t \leq y_t
\]

and all variables are required to be nonnegative.
Assumptions and Comments

In what follows,

- The sequence \( \{\xi_t\} \) is assumed to be IID
- The common distribution of each \( \xi_t \) will be denoted \( \phi \)
- The production function \( f \) is assumed to be increasing and continuous
- Depreciation of capital is not made explicit but can be incorporated into the production function

While many other treatments of the stochastic growth model use \( k_t \) as the state variable, we will use \( y_t \). This will allow us to treat a stochastic model while maintaining only one state variable. We consider alternative states and timing specifications in some of our other lectures.

Optimization

Taking \( y_0 \) as given, the agent wishes to maximize

\[
E \left[ \sum_{t=0}^{\infty} \beta^t u(c_t) \right]
\]

subject to

\[
y_{t+1} = f(y_t - c_t)\xi_{t+1} \quad \text{and} \quad 0 \leq c_t \leq y_t \quad \text{for all} \ t
\]

where

- \( u \) is a bounded, continuous and strictly increasing utility function and
- \( \beta \in (0, 1) \) is a discount factor

In (3.36) we are assuming that the resource constraint (3.34) holds with equality which is reasonable because \( u \) is strictly increasing and no output will be wasted at the optimum.

In summary, the agents aim is to select a path \( c_0, c_1, c_2, \ldots \) for consumption that is

1. nonnegative,
2. feasible in the sense of (3.34),
3. optimal, in the sense that it maximizes (3.35) relative to all other feasible consumption sequences, and
4. adapted, in the sense that the action \( c_t \) depends only on observable outcomes, not future outcomes such as \( \xi_{t+1} \)

In the present context

- \( y_t \) is called the state variable it summarizes the state of the world at the start of each period
- \( c_t \) is called the control variable a value chosen by the agent each period after observing the state
The Policy Function Approach

One way to think about solving this problem is to look for the best policy function. A policy function is a map from past and present observables into current action. We'll be particularly interested in Markov policies, which are maps from the current state $y_t$ into a current action $c_t$.

For dynamic programming problems such as this one (in fact for any Markov decision process), the optimal policy is always a Markov policy.

In other words, the current state $y_t$ provides a sufficient statistic for the history in terms of making an optimal decision today.

This is quite intuitive but if you wish you can find proofs in texts such as [SLP89] (section 4.1). Hereafter we focus on finding the best Markov policy.

In our context, a Markov policy is a function $\sigma : \mathbb{R}_+ \to \mathbb{R}_+$, with the understanding that states are mapped to actions via

$$c_t = \sigma(y_t) \text{ for all } t$$

In what follows, we will call $\sigma$ a feasible consumption policy if it satisfies

$$0 \leq \sigma(y) \leq y \text{ for all } y \in \mathbb{R}_+ \quad (3.37)$$

In other words, a feasible consumption policy is a Markov policy that respects the resource constraint.

The set of all feasible consumption policies will be denoted by $\Sigma$.

Each $\sigma \in \Sigma$ determines a continuous state Markov process $\{y_t\}$ for output via

$$y_{t+1} = f(y_t - \sigma(y_t))x_{t+1}, \quad y_0 \text{ given} \quad (3.38)$$

This is the time path for output when we choose and stick with the policy $\sigma$.

We insert this process into the objective function to get

$$\mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(c_t) \right] = \mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(\sigma(y_t)) \right] \quad (3.39)$$

This is the total expected present value of following policy $\sigma$ forever, given initial income $y_0$.

The aim is to select a policy that makes this number as large as possible.

The next section covers these ideas more formally.
Optimality

The **policy value function** $v_\sigma$ associated with a given policy $\sigma$ is the mapping defined by

$$v_\sigma(y) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(\sigma(y_t)) \right]$$

(3.40)

when $\{y_t\}$ is given by (3.38) with $y_0 = y$.

In other words, it is the lifetime value of following policy $\sigma$ starting at initial condition $y$.

The **value function** is then defined as

$$v^*(y) := \sup_{\sigma \in \Sigma} v_\sigma(y)$$

(3.41)

The value function gives the maximal value that can be obtained from state $y$, after considering all feasible policies.

A policy $\sigma \in \Sigma$ is called **optimal** if it attains the supremum in (3.41) for all $y \in \mathbb{R}_+$.

**The Bellman Equation**

With our assumptions on utility and production function, the value function as defined in (3.41) also satisfies a **Bellman equation**.

For this problem, the Bellman equation takes the form

$$w(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y - c)z) \phi(dz) \right\} \quad (y \in \mathbb{R}_+)$$

(3.42)

This is a **functional equation in** $w$.

The term $\int w(f(y - c)z) \phi(dz)$ can be understood as the expected next period value when

- $w$ is used to measure value
- the state is $y$
- consumption is set to $c$

As shown in **EDTC**, theorem 10.1.11 and a range of other texts

*The value function $v^*$ satisfies the Bellman equation*.

In other words, (3.42) holds when $w = v^*$.

The intuition is that maximal value from a given state can be obtained by optimally trading off

- current reward from a given action, vs
- expected discounted future value of the state resulting from that action.
The Bellman equation is important because it gives us more information about the value function. It also suggests a way of computing the value function, which we discuss below.

**Greedy policies**

The primary importance of the value function is that we can use it to compute optimal policies. The details are as follows:

Given a continuous function $w$ on $\mathbb{R}_+$, we say that $\sigma \in \Sigma$ is $w$-greedy if $\sigma(y)$ is a solution to

$$
\max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y-c)z)\phi(dz) \right\}
$$

(3.43)

for every $y \in \mathbb{R}_+$.

In other words, $\sigma \in \Sigma$ is $w$-greedy if it optimally trades off current and future rewards when $w$ is taken to be the value function.

In our setting, we have the following key result:

*A feasible consumption policy is optimal if and only if it is $v^*$-greedy.*

The intuition is similar to the intuition for the Bellman equation, which was provided after (3.42).

See, for example, theorem 10.1.11 of EDTC.

Hence, once we have a good approximation to $v^*$, we can compute the (approximately) optimal policy by computing the corresponding greedy policy.

The advantage is that we are now solving a much lower dimensional optimization problem.

**The Bellman Operator**

How, then, should we compute the value function?

One way is to use the so-called **Bellman operator** (An operator is a map that sends functions into functions).

The Bellman operator is denoted by $T$ and defined by

$$
Tw(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y-c)z)\phi(dz) \right\} \quad (y \in \mathbb{R}_+)
$$

(3.44)

In other words, $T$ sends the function $w$ into the new function $Tw$ defined (3.44).

By construction, the set of solutions to the Bellman equation (3.42) *exactly coincides with* the set of fixed points of $T$.
For example, if $Tw = w$, then, for any $y \geq 0$,

$$w(y) = Tw(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y-c)z)\phi(dz) \right\}$$

which says precisely that $w$ is a solution to the Bellman equation.

It follows that $v^*$ is a solution to the Bellman equation.

**Review of Theoretical Results**

One can also show that $T$ is a contraction mapping on the set of continuous bounded functions on $\mathbb{R}_+$ under the supremum distance

$$\rho(g, h) = \sup_{y \geq 0} |g(y) - h(y)|$$

See [EDTC], lemma 10.1.18

Hence it has exactly one fixed point in this set, which we know is equal to the value function.

It follows that

- The value function $v^*$ is bounded and continuous
- Starting from any bounded and continuous $w$, the sequence $w, Tw, T^2w, \ldots$ generated by iteratively applying $T$ converges uniformly to $v^*$

This iterative method is called **value function iteration**.

We also know that a feasible policy is optimal if and only if it is $v^*$-greedy.

It’s not too hard to show that a $v^*$-greedy policy exists (see [EDTC], theorem 10.1.11 if you get stuck).

Hence at least one optimal policy exists.

Our problem now is how to compute it.

**Unbounded Utility**

The results stated above assume that the utility function is bounded.

In practice economists often work with unbounded utility functions and so will we.

In the unbounded setting, various optimality theories exist.

Unfortunately, they tend to be case specific, as opposed to valid for a large range of applications.

Nevertheless, their main conclusions are usually in line with those stated for the bounded case just above (as long as we drop the word bounded).

Consult, for example, section 12.2 of [EDTC], [Kam12] or [MdRV10].

**3.8.3 Computation**

Let’s now look at computing the value function and the optimal policy.
Fitted Value Iteration

The first step is to compute the value function by value function iteration

In theory, the algorithm is as follows

1. Begin with a function $w$ an initial condition
2. Solving (3.44), obtain the function $Tw$
3. Unless some stopping condition is satisfied, set $w = Tw$ and go to step 2

This generates the sequence $w, Tw, T^2w, \ldots$

However, there is a problem we must confront before we implement this procedure: The iterates can neither be calculated exactly nor stored on a computer

To see the issue, consider (3.44)

Even if $w$ is a known function, unless $Tw$ can be shown to have some special structure, the only way to store it is to record the value $Tw(y)$ for every $y \in \mathbb{R}_+$

Clearly this is impossible

What we will do instead is use fitted value function iteration

The procedure is to record the value of the function $Tw$ at only finitely many grid points $y_1 < y_2 < \cdots < y_I$ and reconstruct it from this information when required

More precisely, the algorithm will be

1. Begin with an array of values $\{w_1, \ldots, w_I\}$ representing the values of some initial function $w$ on the grid points $\{y_1, \ldots, y_I\}$
2. Build a function $\hat{w}$ on the state space $\mathbb{R}_+$ by interpolation or approximation, based on these data points
3. Obtain and record the value $T\hat{w}(y_i)$ on each grid point $y_i$ by repeatedly solving (3.44)
4. Unless some stopping condition is satisfied, set $\{w_1, \ldots, w_I\} = \{T\hat{w}(y_1), \ldots, T\hat{w}(y_I)\}$ and go to step 2

How should we go about step 2?

This is a problem of function approximation, and there are many ways to approach it

Whats important here is that the function approximation scheme must not only produce a good approximation to $Tw$, but also combine well with the broader iteration algorithm described above

One good choice from both respects is continuous piecewise linear interpolation (see this paper for further discussion)

The next figure illustrates piecewise linear interpolation of an arbitrary function on grid points 0, 0.2, 0.4, 0.6, 0.8, 1

```python
using PyPlot
using QuantEcon

f(x) = 2.*cos.(6x) .+ sin.(14x) .+ 2.5
c_grid = 0:.2:1
```
Another advantage of piecewise linear interpolation is that it preserves useful shape properties such as monotonicity and concavity / convexity

**The Bellman Operator**

Here is a function that implements the Bellman operator using linear interpolation:

```julia
#=
@authors : Spencer Lyon, John Stachurski
=#
using Optim

f_grid = linspace(0, 1, 150)
Af = LinInterp(c_grid, f(c_grid))

fig, ax = plt[:,subplots](figsize=(10, 6))
ax[:,set](xlim=(0, 1), ylim=(0, 6))
ax[:,plot](f_grid, f(f_grid), "b-", lw=2, alpha=0.8, label="true function")
ax[:,plot](f_grid, Af.(f_grid), "g-", lw=2, alpha=0.8, label="linear approximation")
ax[:,vlines](c_grid, c_grid * 0, f(c_grid), linestyle="dashed", alpha=0.5)
ax[:,legend](loc="upper center")
```
The approximate Bellman operator, which computes and returns the updated value function Tw on the grid points. An array to store the new set of values Tw is optionally supplied (to avoid having to allocate new arrays at each iteration). If supplied, any existing data in Tw will be overwritten.

#### Arguments

`w` : Vector  
  The value of the input function on different grid points  
`grid` : Vector  
  The set of grid points  
`β` : AbstractFloat  
  The discount factor  
`u` : Function  
  The utility function  
`f` : Function  
  The production function  
`shocks` : Vector  
  An array of draws from the shock, for Monte Carlo integration (to compute expectations).  
`Tw` : Vector, optional (default=similar(w))  
  Array to write output values to  
`compute_policy` : Bool, optional (default=false)  
  Whether or not to compute policy function

```julia
function bellman_operator(w::Vector,
  grid::Vector,
  β::AbstractFloat,
  u::Function,
  f::Function,
  shocks::Vector,
  Tw::Vector = similar(w);
  compute_policy::Bool = false)

  # === Apply linear interpolation to w === #
  w_func = LinInterp(grid, w)

  if compute_policy
    σ = similar(w)
  end

  # == set Tw[i] = max_c ( u(c) + β E w(f(y - c) z)) == #
  for (i, y) in enumerate(grid)
    objective(c) = -u(c) - β * mean(w_func.(f(y - c).*shocks))
    res = optimize(objective, 1e-10, y)
    if compute_policy
      σ[i] = res.minimizer
  end
  Tw = σ

  return Tw
```

The arguments to `bellman_operator` are described in the docstring to the function.

Notice that the expectation in (3.44) is computed via Monte Carlo, using the approximation

\[ \int w(f(y - c)z)\phi(dz) \approx \frac{1}{n} \sum_{i=1}^{n} w(f(y - c)\xi_i) \]

where \( \{\xi_i\}_{i=1}^{n} \) are IID draws from \( \phi \).

Monte Carlo is not always the most efficient way to compute integrals numerically but it does have some theoretical advantages in the present setting.

(For example, it preserves the contraction mapping property of the Bellman operator see, e.g., [PalS13])

**An Example**

Lets test out our operator when

- \( f(k) = k^\alpha \)
- \( u(c) = \ln c \)
- \( \phi \) is the distribution of \( \exp(\mu + \sigma \zeta) \) when \( \zeta \) is standard normal.

As is well-known (see [LS18], section 3.1.2), for this particular problem an exact analytical solution is available, with

\[ v^*(y) = \frac{\ln(1 - \alpha \beta)}{1 - \beta} + \frac{\mu + \alpha \ln(\alpha \beta)}{1 - \alpha} \left[ \frac{1}{1 - \beta} - \frac{1}{1 - \alpha \beta} \right] + \frac{1}{1 - \alpha \beta} \ln y \]  

(3.45)

The optimal consumption policy is

\[ \sigma^*(y) = (1 - \alpha \beta)y \]

Lets code this up now so we can test against it below.
s = 0.1

c1 = \log(1 - \alpha \beta) / (1 - \beta)
c2 = (\mu + \alpha \log(\alpha \beta)) / (1 - \alpha)
c3 = 1 / (1 - \beta)
c4 = 1 / (1 - \alpha \beta)

# Utility
u(c) = \log(c)
u'(c) = 1 / c

# Deterministic part of production function
f(k) = k^\alpha
f'(k) = \alpha \cdot k^{\alpha - 1}

# True optimal policy
c^*(y) = (1 - \alpha \beta) \cdot y

# True value function
v^*(y) = c1 + c2 \cdot (c3 - c4) + c4 \cdot \log(y)

A First Test

To test our code, we want to see if we can replicate the analytical solution numerically, using fitted value function iteration

We need a grid and some shock draws for Monte Carlo integration

```
grid_max = 4 # Largest grid point
grid_size = 200 # Number of grid points
shock_size = 250 # Number of shock draws in Monte Carlo integral

grid_y = collect(linspace(1e-5, grid_max, grid_size))
shocks = exp.((\mu + s) \cdot \text{randn}(\text{shock_size}))
```

Now let's do some tests

As one preliminary test, let's see what happens when we apply our Bellman operator to the exact solution \( v^* \)

In theory, the resulting function should again be \( v^* \)

In practice we expect some small numerical error

```
w = bellman_operator(v^*(grid_y),
grid_y, \beta,
log, k -> k^\alpha,
shocks)
```
The two functions are essentially indistinguishable, so we are off to a good start.

Now let's have a look at iterating with the Bellman operator, starting off from an arbitrary initial condition.

The initial condition well start with is $w(y) = 5 \ln(y)$

```python
w = 5 * log.(grid_y)  # An initial condition -- fairly arbitrary
n = 35
fig, ax = subplots(figsize=(9, 6))
ax[:set_ylim](-50, 10)
ax[:set_xlim](minimum(grid_y), maximum(grid_y))
lb = "initial condition"
jet = ColorMap("jet")
ax[:plot](grid_y, w, color=jet(0), lw=2, alpha=0.6, label=lb)
for i in 1:n
    w = bellman_operator(w, grid_y,
```
The figure shows

1. the first 36 functions generated by the fitted value function iteration algorithm, with hotter colors given to higher iterates
2. the true value function $v^*$ drawn in black

The sequence of iterates converges towards $v^*$

We are clearly getting closer

We can write a function that iterates until the difference is below a particular tolerance level
function solve_optgrowth(initial_w;  
    tol::AbstractFloat=1e-6,  
    max_iter::Integer=500)  

    w = initial_w  # Set initial condition  
    error = tol + 1  
    i = 0  

    # == Create storage array for bellman_operator. Reduces memory  
    # allocation and speeds code up == #  
    Tw = similar(grid_y)  

    # Iterate to find solution  
    while (error > tol) && (i < max_iter)  
        w_new = bellman_operator(w,  
            grid_y,  
            β,  
            log,  
            k -> k^α,  
            shocks)  
        error = maximum(abs, w_new - w)  
        w = w_new  
        i += 1  
    end  

    return w  
end

We can check our result by plotting it against the true value

initial_w = 5 * log.(grid_y)  
v_star_approx = solve_optgrowth(initial_w)  

fig, ax = subplots(figsize=(9, 5))  
a[:set_ylim](-35, -24)  
a[:plot](grid_y, v_star_approx, lw=2, alpha=0.6, label="approximate value")  
a[:plot](grid_y, v_star.(grid_y), lw=2, alpha=0.6, label="true value function")  
a[:legend](loc="lower right")  
show()
Alternatively, we can use QuantEcon's `compute_fixed_point` function to converge to \( v^* \)

```julia
import QuantEcon: compute_fixed_point

Tw = similar(grid_y)
initial_w = 5 * log.(grid_y)

bellman_operator(w) = bellman_operator(w,
                                 grid_y,
                                 \beta,
                                 log,
                                 k -> k^\alpha,
                                 shocks)

v_star_approx = compute_fixed_point(bellman_operator,
                                     initial_w,
                                     max_iter=500,
                                     verbose=2,
                                     print_skip=10,
                                     err_tol=1e-5)
```

Here is the output:

```
Compute iterate 10 with error 0.709153897728406
Compute iterate 20 with error 0.4709584432889145
Compute iterate 30 with error 0.3131085083453158
Compute iterate 40 with error 0.20816475495214704
Compute iterate 50 with error 0.13839472275577336
Compute iterate 60 with error 0.0920093283702069
Compute iterate 70 with error 0.061170804294146564
```
Compute iterate 80 with error 0.04066834706496181
Compute iterate 90 with error 0.027037644380378367
Compute iterate 100 with error 0.01797550838462314
Compute iterate 110 with error 0.011950704623018282
Compute iterate 120 with error 0.00794521845620011
Compute iterate 130 with error 0.005282240529794535
Compute iterate 140 with error 0.0035118058954915273
Compute iterate 150 with error 0.0023347631704915273
Compute iterate 160 with error 0.001552226753144481
Compute iterate 170 with error 0.0010319710041208907
Compute iterate 180 with error 0.0006860880028760619
Compute iterate 190 with error 0.0004561336956570017
Compute iterate 200 with error 0.00030325256715002524
Compute iterate 210 with error 0.00020161220411551994
Compute iterate 220 with error 0.0001340383727708172
Compute iterate 230 with error 8.91308481529545e-5
Compute iterate 240 with error 5.9245287411500258e-5
Compute iterate 250 with error 3.938820087867616e-5
Compute iterate 260 with error 2.6186561168373146e-5
Compute iterate 270 with error 1.7409680953761608e-5
Compute iterate 280 with error 1.1574523867352582e-5
Converged in 284 steps

Lets have a look at the result

```python
fig, ax = subplots(figsize=(9, 5))
ax[0].set_ylim(-35, -24)
ax[0].plot(grid_y, v_star_approx, lw=2, alpha=0.6, label="approximate value function")
ax[0].plot(grid_y, v_star.(grid_y), lw=2, alpha=0.6, label="true value function")
ax[0].legend(loc="lower right")
show()
```
The figure shows that we are pretty much on the money.

**The Policy Function**

To compute an approximate optimal policy, we take the approximate value function we just calculated and then compute the corresponding greedy policy.

The next figure compares the result to the exact solution, which, as mentioned above, is $\sigma(y) = (1 - \alpha \beta)y$.

```python
Tw, sigma = bellman_operator(v_star_approx, 
grid_y, 
\beta, 
log, 
k -> k^\alpha, 
shocks; 
compute_policy=True)

cstar = (1 - \alpha * \beta) * grid_y

fig, ax = subplots(figsize=(9, 5))
ax.plot(grid_y, sigma, lw=2, alpha=0.6, label="approximate policy function")
ax.plot(grid_y, cstar, lw=2, alpha=0.6, label="true policy function")
ax.legend(loc="lower right")
show()
```
The figure shows that we’ve done a good job in this instance of approximating the true policy.

### 3.8.4 Exercises

#### Exercise 1

Once an optimal consumption policy \( \sigma \) is given, income follows (3.38)

The next figure shows a simulation of 100 elements of this sequence for three different discount factors (and hence three different policies).
In each sequence, the initial condition is $y_0 = 0.1$

The discount factors are $\text{discount\_factors} = (0.8, 0.9, 0.98)$

We have also dialed down the shocks a bit

```julia
    s = 0.05
    shocks = exp.(μ + s * randn(shock_size))
```

Otherwise, the parameters and primitives are the same as the log linear model discussed earlier in the lecture

Notice that more patient agents typically have higher wealth

Replicate the figure modulo randomness

### 3.8.5 Solutions

**Exercise 1**

Heres one solution (assuming as usual that youve executed everything above)

```julia
""
Compute a time series given consumption policy $σ$.
""
function simulate_og(σ, y0 = 0.1, ts_length=100)
```

```julia
y = Array{Float64}(ts_length)
ξ = randn(ts_length-1)
y[1] = y0
for t in 1:(ts_length-1)
    y[t+1] = (y[t] - σ(y[t]))^α * exp(μ + s * ξ[t])
end
return y
end

fig, ax = subplots(figsize=(9, 6))

for β in (0.9, 0.94, 0.98)
    Tw = similar(grid_y)
    initial_w = 5 * log.(grid_y)
    v_star_approx = compute_fixed_point(bellman_operator,
                                          initial_w,
                                          max_iter=50,
                                          verbose=0,
                                          print_skip=10,
                                          err_tol=1e-5)

    Tw, σ = bellman_operator(v_star_approx,
                              grid_y,
                              β,
                              log,
                              k -> k^α,
                              shocks,
                              compute_policy=true)

    σ_func = LinInterp(grid_y, σ)
y = simulate_og(σ_func)
ax[plot](y, lw=2, alpha=0.6, label="β = $β")
end

ax[legend](loc="lower right")
show()
```
3.9 Optimal Growth II: Time Iteration

In this lecture we continue our earlier study of the stochastic optimal growth model.
In that lecture we solved the associated discounted dynamic programming problem using value function iteration.

The beauty of this technique is its broad applicability. With numerical problems, however, we can often attain higher efficiency in specific applications by deriving methods that are carefully tailored to the application at hand.

The stochastic optimal growth model has plenty of structure to exploit for this purpose, especially when we adopt some concavity and smoothness assumptions over primitives.

We'll use this structure to obtain an Euler equation based method that's more efficient than value function iteration for this and some other closely related applications.

In a subsequent lecture we'll see that the numerical implementation part of the Euler equation method can be further adjusted to obtain even more efficiency.

### 3.9.2 The Euler Equation

Let's take the model set out in the stochastic growth model lecture and add the assumptions that

1. $u$ and $f$ are continuously differentiable and strictly concave
2. $f(0) = 0$
3. $\lim_{c \to 0} u'(c) = \infty$ and $\lim_{c \to \infty} u'(c) = 0$
4. $\lim_{k \to 0} f'(k) = \infty$ and $\lim_{k \to \infty} f'(k) = 0$

The last two conditions are usually called Inada conditions.

Recall the Bellman equation

$$v^*(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y - c)z)\phi(dz) \right\} \quad \text{for all } y \in \mathbb{R}_+ \quad (3.46)$$

Let the optimal consumption policy be denoted by $c^*$.

We know that $c^*$ is a $v^*$ greedy policy, so that $c^*(y)$ is the maximizer in $(3.46)$.

The conditions above imply that

- $c^*$ is the unique optimal policy for the stochastic optimal growth model.
- the optimal policy is continuous, strictly increasing and also interior, in the sense that $0 < c^*(y) < y$ for all strictly positive $y$.
- the value function is strictly concave and continuously differentiable, with

$$\left(v^*(y)\right)' = u'(c^*(y)) := (u' \circ c^*)(y) \quad (3.47)$$

The last result is called the envelope condition due to its relationship with the envelope theorem.

To see why $(3.47)$ might be valid, write the Bellman equation in the equivalent form

$$v^*(y) = \max_{0 \leq k \leq y} \left\{ u(y - k) + \beta \int v^*(f(k)z)\phi(dz) \right\},$$

\[\text{Chapter 3. Dynamic Programming}\]
differentiate naively with respect to $y$, and then evaluate at the optimum

Section 12.1 of EDTC contains full proofs of these results, and closely related discussions can be found in many other texts

Differentiability of the value function and iteriority of the optimal policy imply that optimal consumption satisfies the first order condition associated with (3.46), which is

$$u'(c^*(y)) = \beta \int (u^*)(f(y - c^*(y))z)f'(y - c^*(y))z\phi(dz) \quad (3.48)$$

Combining (3.47) and the first-order condition (3.48) gives the famous Euler equation

$$(u' \circ c^*)(y) = \beta \int (u' \circ c^*)(f(y - c^*(y))z)f'(y - c^*(y))z\phi(dz) \quad (3.49)$$

We can think of the Euler equation as a functional equation

$$(u' \circ \sigma)(y) = \beta \int (u' \circ \sigma)(f(y - \sigma(y))z)f'(y - \sigma(y))z\phi(dz) \quad (3.50)$$

over interior consumption policies $\sigma$, one solution of which is the optimal policy $c^*$

Our aim is to solve the functional equation (3.50) and hence obtain $c^*$

**The Coleman Operator**

Recall the Bellman operator

$$Tw(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y - c)z)\phi(dz) \right\} \quad (3.51)$$

Just as we introduced the Bellman operator to solve the Bellman equation, we will now introduce an operator over policies to help us solve the Euler equation

This operator $K$ will act on the set of all $\sigma \in \Sigma$ that are continuous, strictly increasing and interior (i.e., $0 < \sigma(y) < y$ for all strictly positive $y$)

Henceforth we denote this set of policies by $P$

1. The operator $K$ takes as its argument a $\sigma \in P$ and
2. returns a new function $K\sigma$, where $K\sigma(y)$ is the $c \in (0, y)$ that solves

$$u'(c) = \beta \int (u' \circ \sigma)(f(y - c)z)f'(y - c)z\phi(dz) \quad (3.52)$$

We call this operator the Coleman operator to acknowledge the work of [Col90] (although many people have studied this and other closely related iterative techniques)
In essence, $K\sigma$ is the consumption policy that the Euler equation tells you to choose today when your future consumption policy is $\sigma$

The important thing to note about $K$ is that, by construction, its fixed points coincide with solutions to the functional equation (3.50)

In particular, the optimal policy $c^*$ is a fixed point

Indeed, for fixed $y$, the value $Kc^*(y)$ is the $c$ that solves

$$u'(c) = \beta \int (u' \circ c^*)(f(y - c)z)f'(y - c)z\phi(dz)$$

In view of the Euler equation, this is exactly $c^*(y)$

**Is the Coleman Operator Well Defined?**

In particular, is there always a unique $c \in (0, y)$ that solves (3.52)?

The answer is yes, under our assumptions

For any $\sigma \in \mathcal{P}$, the right side of (3.52)

- is continuous and strictly increasing in $c$ on $(0, y)$
- diverges to $+\infty$ as $c \uparrow y$

The left side of (3.52)

- is continuous and strictly decreasing in $c$ on $(0, y)$
- diverges to $+\infty$ as $c \downarrow 0$

Sketching these curves and using the information above will convince you that they cross exactly once as $c$ ranges over $(0, y)$

With a bit more analysis, one can show in addition that $K\sigma \in \mathcal{P}$ whenever $\sigma \in \mathcal{P}$

**3.9.3 Comparison with Value Function Iteration**

How does Euler equation time iteration compare with value function iteration?

Both can be used to compute the optimal policy, but is one faster or more accurate?

There are two parts to this story

First, on a theoretical level, the two methods are essentially isomorphic

In particular, they converge at the same rate

Well prove this in just a moment

The other side to the story is the speed of the numerical implementation

It turns out that, once we actually implement these two routines, time iteration is faster and more accurate than value function iteration

More on this below
Equivalent Dynamics

Lets talk about the theory first

To explain the connection between the two algorithms, it helps to understand the notion of equivalent dynamics

(This concept is very helpful in many other contexts as well)

Suppose that we have a function $g : X \to X$ where $X$ is a given set

The pair $(X, g)$ is sometimes called a **dynamical system** and we associate it with trajectories of the form

$$x_{t+1} = g(x_t), \quad x_0 \text{ given}$$

Equivalently, $x_t = g^t(x_0)$, where $g$ is the $t$-th composition of $g$ with itself

Here is the picture

$$
\begin{array}{cccccc}
X_0 & \xrightarrow{g} & g(x_0) & \xrightarrow{g} & g^2(x_0) & \xrightarrow{g} & g^3(x_0) & \xrightarrow{g} & \cdots \\
\end{array}
$$

Now let another function $h : Y \to Y$ where $Y$ is another set

Suppose further that

- there exists a bijection $\tau$ from $X$ to $Y$
- the two functions **commute** under $\tau$, which is to say that $\tau(g(x)) = h(\tau(x))$ for all $x \in X$

The last statement can be written more simply as

$$\tau \circ g = h \circ \tau$$

or, by applying $\tau^{-1}$ to both sides

$$g = \tau^{-1} \circ h \circ \tau \quad (3.53)$$

Here is a commutative diagram that illustrates

$$
\begin{array}{ccc}
X & \xrightarrow{g} & X \\
\downarrow{\tau} & & \uparrow{\tau^{-1}} \\
Y & \xrightarrow{h} & Y
\end{array}
$$
Heres a similar figure that traces out the action of the maps on a point \( x \in X \)

\[
\begin{align*}
  x & \xrightarrow{g} g(x) \\
  \tau \downarrow & \uparrow \tau^{-1} \\
  \tau(x) & \xrightarrow{h} h(\tau(x))
\end{align*}
\]

Now, its easy to check from (3.53) that \( g^2 = \tau^{-1} \circ h^2 \circ \tau \) holds

In fact, if you like proofs by induction, you wont have trouble showing that

\[ g^n = \tau^{-1} \circ h^n \circ \tau \]

is valid for all \( n \)

What does this tell us?

It tells us that the following are equivalent

- iterate \( n \) times with \( g \), starting at \( x \)
- shift \( x \) to \( Y \) using \( \tau \), iterate \( n \) times with \( h \) starting at \( \tau(x) \), and shift the result \( h^n(\tau(x)) \) back to \( X \) using \( \tau^{-1} \)

We end up with exactly the same object

**Back to Economics**

Have you guessed where this is leading?

What were going to show now is that the operators \( T \) and \( K \) commute under a certain bijection

The implication is that they have exactly the same rate of convergence

To make life a little easier, well assume in the following analysis (although not always in our applications) that \( u(0) = 0 \)

**A Bijection**

Let \( \mathcal{V} \) be all strictly concave, continuously differentiable functions \( v \) mapping \( \mathbb{R}_+ \) to itself and satisfying \( v(0) = 0 \) and \( v'(y) > u'(y) \) for all positive \( y \)

For \( v \in \mathcal{V} \) let

\[ Mv := h \circ v' \quad \text{where} \quad h := (u')^{-1} \]

Although we omit details, \( \sigma := Mv \) is actually the unique \( v \)-greedy policy
• See proposition 12.1.18 of **EDTC**

It turns out that $M$ is a bijection from $\mathcal{V}$ to $\mathcal{P}$

A (solved) exercise below asks you to confirm this

**Commutative Operators**

It is an additional solved exercise (see below) to show that $T$ and $K$ commute under $M$, in the sense that

$$M \circ T = K \circ M$$

(3.54)

In view of the preceding discussion, this implies that

$$T^n = M^{-1} \circ K^n \circ M$$

Hence, $T$ and $K$ converge at exactly the same rate!

**3.9.4 Implementation**

Weve just shown that the operators $T$ and $K$ have the same rate of convergence

However, it turns out that, once numerical approximation is taken into account, significant differences arises

In particular, the image of policy functions under $K$ can be calculated faster and with greater accuracy than the image of value functions under $T$

Our intuition for this result is that

• the Coleman operator exploits more information because it uses first order and envelope conditions

• policy functions generally have less curvature than value functions, and hence admit more accurate approximations based on grid point information

**The Operator**

Here is some code that implements the Coleman operator

```julia
#= Author: Shunsuke Hori

#=

using QuantEcon

"""
g: input policy function
grid: grid points
```
\(\beta\): discount factor
\(u'_\text{prime}\): derivative of utility function
\(f\): production function
\(f'_\text{prime}\): derivative of production function
\text{shocks}: shock draws, used for Monte Carlo integration to compute expectation
\(K_g\): output value is stored

```
function coleman_operator!(g::AbstractVector,
    grid::AbstractVector,
    \(\beta\)::AbstractFloat,
    u_prime::Function,
    f::Function,
    f_prime::Function,
    shocks::AbstractVector,
    Kg::AbstractVector=similar(g))

    # This function requires the container of the output value as argument \(K_g\)

    # Construct linear interpolation object #
    g_func = LinInterp(grid, g)

    # solve for updated consumption value #
    for (i, y) in enumerate(grid)
        function h(c)
            vals = u_prime.(g_func.(f(y - c) \times shocks)) \times f_prime(y - c) \times shocks
            return u_prime(c) - \(\beta\) \times mean(vals)
        end
        Kg[i] = brent(h, 1e-10, y - 1e-10)
    end
    return Kg
end

# The following function does NOT require the container of the output value,
# as argument
function coleman_operator(g::AbstractVector,
    grid::AbstractVector,
    \(\beta\)::AbstractFloat,
    u_prime::Function,
    f::Function,
    f_prime::Function,
    shocks::AbstractVector)

    return coleman_operator!(g, grid, \(\beta\), u_prime,
        f, f_prime, shocks, similar(g))
end
```

It has some similarities to the code for the Bellman operator in our \textit{optimal growth lecture}

For example, it evaluates integrals by Monte Carlo and approximates functions using linear interpolation

Here's that Bellman operator code again, which needs to be executed because well use it in some tests below
## 3.9. Optimal Growth II: Time Iteration

```julia
#=
@authors : Spencer Lyon, John Stachurski
=#

```using` Optim

```

""
The approximate Bellman operator, which computes and returns the updated value function $T_w$ on the grid points. An array to store the new set of values $T_w$ is optionally supplied (to avoid having to allocate new arrays at each iteration). If supplied, any existing data in $T_w$ will be overwritten.

### Arguments

- `'w'`: `Vector`
  - The value of the input function on different grid points
- `'grid'`: `Vector`
  - The set of grid points
- `'β'`: `AbstractFloat`
  - The discount factor
- `'u'`: `Function`
  - The utility function
- `'f'`: `Function`
  - The production function
- `'shocks'`: `Vector`
  - An array of draws from the shock, for Monte Carlo integration (to compute expectations).
- `'Tw'`: `Vector`, optional (default=`similar(w)`) Array to write output values to
- `'compute_policy'`: `Bool`, optional (default=`false`) Whether or not to compute policy function

""

```julia
function bellman_operator(w::Vector,
    grid::Vector,
    β::AbstractFloat,
    u::Function,
    f::Function,
    shocks::Vector,
    Tw::Vector = similar(w);
    compute_policy::Bool = false)

    # === Apply linear interpolation to w === #
    w_func = LinInterp(grid, w)

    if compute_policy
        σ = similar(w)
    end

```
# == set Tw[i] = max_c { u(c) + \beta E w(f(y - c) z)} == #

for (i, y) in enumerate(grid)
    objective(c) = -u(c) - \beta * mean(w_func.(f(y - c) .* shocks))
    res = optimize(objective, 1e-10, y)

    if compute_policy
        \sigma[i] = res.minimizer
    end
    Tw[i] = - res.minimum
end

if compute_policy
    return Tw, \sigma
else
    return Tw
end
end

Testing on the Log / Cobb–Douglas case

As we did for value function iteration, lets start by testing our method in the presence of a model that does have an analytical solution.

Here's a struct containing data from the log-linear growth model we used in the value function iteration lecture.

```julia
# =

Author: Shunsuke Hori
=

struct Model{TF <: AbstractFloat, TR <: Real, TI <: Integer}
    \alpha::TR   # Productivity parameter
    \beta::TF   # Discount factor
    \gamma::TR  # Risk aversion
    \mu::TR     # First parameter in lognorm(\mu, \sigma)
    \sigma::TR  # Second parameter in lognorm(\mu, \sigma)
    grid_min::TR # Smallest grid point
    grid_max::TR # Largest grid point
    grid_size::TI # Number of grid points
    u::Function # Utility function
    u_prime::Function # Derivative of utility function
    f::Function # Production function
    f_prime::Function # Derivative of production function
    grid::Vector{TR} # Grid
end

""
construct Model instance using the information of parameters and functional form
```

Chapter 3. Dynamic Programming
arguments: see above
return: Model type instance

function Model(; α::Real=0.65,  # Productivity parameter
β::AbstractFloat=0.95,   # Discount factor
γ::Real=1.0,            # risk aversion
μ::Real=0.0,            # First parameter in
% lognorm(μ, σ)
s::Real=0.1,            # Second parameter in
% lognorm(μ, σ)
grid_min::Real=1e-6,    # Smallest grid point
grid_max::Real=4.0,     # Largest grid point
grid_size::Integer=200, # Number of grid points
u::Function= c -> (c^(1-γ)-1)/(1-γ), # utility function
u_prime::Function = c -> c^(-γ),        # u'
f::Function = k -> k*α,               # production function
f_prime::Function = k -> α*k^(α-1)    # f'
)

grid = collect(linspace(grid_min, grid_max, grid_size))

if γ == 1
  # when γ=1, log utility is assigned
  u_log(c) = log(c)
  m = Model(α, β, γ, μ, s, grid_min, grid_max,
            grid_size, u_log, u_prime, f, f_prime, grid)
else
  m = Model(α, β, γ, μ, s, grid_min, grid_max,
            grid_size, u, u_prime, f, f_prime, grid)
end
return m
end

Next we generate an instance

m = Model(γ=1.0)  # model instance with specific parameter

We also need some shock draws for Monte Carlo integration

shock_size = 250  # Number of shock draws in Monte Carlo integral
shocks = collect(exp.(m.μ + m.s * randn(shock_size)))  # generate shocks

As a preliminary test, lets see if $Kc^* = c^*$, as implied by the theory

using PyPlot

""
This function plots a specified policy function and map of the policy function
If they are same, the policy function is true

3.9. Optimal Growth II: Time Iteration
m: instance of Model type, storing all parameters
c: shocks: shocks for Monte Carlo integration
c_star: true policy function

```
function verify_true_policy(m::Model,
    shocks::AbstractVector,
    c_star::AbstractVector)
    # Compute (Kc^*)
    c_star_new = coleman_operator(c_star, m.beta, m.f_prime, m.f_prime, shocks)

    # Plot c^* and Kc^* #
    fig, ax = subplots()
    ax[1].plot(m.grid, c_star, label=$"optimal policy $c^*\$"
    ax[1].plot(m.grid, c_star_new, label=$"Kc^*\$"
    ax[1].legend(loc="upper left")
end

```
c_star = (1 - m.alpha * m.beta) * m.grid # True policy (c^*)
verify_true_policy(m, shocks, c_star)

Here is the result:
We can't really distinguish the two plots, so we are looking good, at least for this test.

Next let's try iterating from an arbitrary initial condition and see if we converge towards $c^*$.

The initial condition we will use is the one that eats the whole pie: $c(y) = y$

```julia
function check_convergence(m::Model,
    shocks::AbstractVector,
    c_star::AbstractVector,
    g_init::AbstractVector;
    n_iter::Integer=15)

    This function plots the resulting policy function of each iteration and compares it with the true one.

    m: instance of Model type, storing all parameters
    shocks: shocks for Monte Carlo integration
    c_star: true policy function
    n_iter: number of iterations
    g_init: initial policy function

end
```
We see that the policy has converged nicely, in only a few steps.
Now let's compare the accuracy of iteration using the Coleman and Bellman operators.

Well generate

1. $K^nc$ where $c(y) = y$
2. $(M \circ T^n \circ M^{-1})c$ where $c(y) = y$

In each case, we'll compare the resulting policy to $c^*$.

The theory on equivalent dynamics says we will get the same policy function and hence the same errors.

But in fact, we expect the first method to be more accurate for reasons discussed above.

```
function iterate_updating(func::Function,
    arg_init::AbstractVector;
    sim_length::Integer=20)

    arg = arg_init;
    for i=1:sim_length
        new_arg = func(arg)
        arg = new_arg
    end
    return arg
end

function compare_error(m::Model,
    shocks::AbstractVector,
    g_init::AbstractVector,
    w_init::AbstractVector;
    sim_length::Integer=20)

    g, w = g_init, w_init
    # two functions for simplification
    bellman_single_arg(w) = bellman_operator(w, m.grid, m.β, m.u,
```

---

3.9. Optimal Growth II: Time Iteration
coleman_single_arg(g) = coleman_operator(g, m.grid, m.β, m.u_prime, m.f, m.f_prime, shocks)

g = iterate_updating(coleman_single_arg, m.grid, sim_length=20)
w = iterate_updating(bellman_single_arg, m.u.(m.grid), sim_length=20)
new_w, vf_g = bellman_operator(w, m.grid, m.β, m.u, m.f, shocks, compute_policy=true)

fig, ax = subplots()
ax[:plot](m.grid, 0 * m.grid, "k-", lw=1)
ax[:plot](m.grid, pf_error, lw=2, alpha=0.6, label="policy iteration error")
ax[:plot](m.grid, vf_error, lw=2, alpha=0.6, label="value iteration error")
ax[:legend](loc="lower left")
end

compare_error(m, shocks, m.grid, m.u.(m.grid), sim_length=20)

Here's the result, which shows the errors in each case.
As you can see, time iteration is much more accurate for a given number of iterations.

### 3.9.5 Exercises

**Exercise 1**

Show that (3.54) is valid. In particular,

- Let \( v \) be strictly concave and continuously differentiable on \((0, \infty)\)
- Fix \( y \in (0, \infty) \) and show that \( MTv(y) = KMv(y) \)

**Exercise 2**

Show that \( M \) is a bijection from \( \mathcal{V} \) to \( \mathcal{P} \)
Exercise 3

Consider the same model as above but with the CRRA utility function

\[ u(c) = \frac{c^{1-\gamma} - 1}{1-\gamma} \]

Iterate 20 times with Bellman iteration and Euler equation time iteration

- start time iteration from \( c(y) = y \)
- start value function iteration from \( v(y) = u(y) \)
- set \( \gamma = 1.5 \)

Compare the resulting policies and check that they are close

Exercise 4

Do the same exercise, but now, rather than plotting results, time how long 20 iterations takes in each case

3.9.6 Solutions

Solution to Exercise 1

Let \( T, K, M, v \) and \( y \) be as stated in the exercise

Using the envelope theorem, one can show that \((Tv)'(y) = u'(c(y))\) where \( c(y) \) solves

\[
 u'(c(y)) = \beta \int v'(f(y - c(y))) f'(y - c(y)) z \phi(dz)
\]  (3.55)

Hence \( MTv(y) = (u')^{-1}(u'(c(y))) = c(y) \)

On the other hand, \( KMv(y) \) is the \( c(y) \) that solves

\[
 u'(c(y)) = \beta \int (u' \circ (Mv))(f(y - c(y))) f'(y - c(y)) z \phi(dz)
 = \beta \int (u' \circ ((u')^{-1} \circ v'))(f(y - c(y))) f'(y - c(y)) z \phi(dz)
 = \beta \int v'(f(y - c(y))) f'(y - c(y)) z \phi(dz)
\]

We see that \( c(y) \) is the same in each case
Solution to Exercise 2

We need to show that $M$ is a bijection from $\mathcal{V}$ to $\mathcal{P}$

To see this, first observe that, in view of our assumptions above, $u'$ is a strictly decreasing continuous bijection from $(0, \infty)$ to itself

It follows that $h$ has the same properties

Moreover, for fixed $v \in \mathcal{V}$, the derivative $v'$ is a continuous, strictly decreasing function

Hence, for fixed $v \in \mathcal{V}$, the map $Mv = h \circ v'$ is strictly increasing and continuous, taking values in $(0, \infty)$

Moreover, interiority holds because $v'$ strictly dominates $u'$, implying that

$$(Mv)(y) = h(v'(y)) < h(u'(y)) = y$$

In particular, $\sigma(y) := (Mv)(y)$ is an element of $\mathcal{P}$

To see that each $\sigma \in \mathcal{P}$ has a preimage $v \in \mathcal{V}$ with $Mv = \sigma$, fix any $\sigma \in \mathcal{P}$

Let $v(y) := \int_0^y u'(\sigma(x))dx$ with $v(0) = 0$

With a small amount of effort you will be able to show that $v \in \mathcal{V}$ and $Mv = \sigma$

Its also true that $M$ is one-to-one on $\mathcal{V}$

To see this, suppose that $v$ and $w$ are elements of $\mathcal{V}$ satisfying $Mv = Mw$

Then $v(0) = w(0) = 0$ and $v' = w'$ on $(0, \infty)$

The fundamental theorem of calculus then implies that $v = w$ on $\mathbb{R}_+$

Solution to Exercise 3

Here is the code, which will execute if you've run all the code above

```julia
# Model instance with risk aversion = 1.5
# others are same as the previous instance
m_ex = Model(γ=1.5)

""" This function conducts the analysis of exercise 2
m: instance of Model type, storing all parameters
shocks: shocks for Monte Carlo integration
g_init: initial policy
w_init: initial value
sim_length: number of iterations
"""

function exercise2(m::Model,
                  shocks::AbstractVector,
                  g_init::AbstractVector=m.grid,
                  w_init::AbstractVector=m.u.(m.grid);
```
```
sim_length::Integer=20)

# initial policy and value
g, w = g_init, w_init
# iteration
bellman_single_arg(w) = bellman_operator(w, m.grid, m.β, m.u,
m.f, shocks)
coleman_single_arg(g) = coleman_operator(g, m.grid, m.β, m.u_prime,
m.f, m.f_prime, shocks)

# iteration
bellman_single_arg(w) = bellman_operator(w, m.grid, m.β, m.u,
m.f, shocks)
coleman_single_arg(g) = coleman_operator(g, m.grid, m.β, m.u_prime,
m.f, m.f_prime, shocks)

# iteration
bellman_single_arg(w) = bellman_operator(w, m.grid, m.β, m.u,
m.f, shocks)
coleman_single_arg(g) = coleman_operator(g, m.grid, m.β, m.u_prime,
m.f, m.f_prime, shocks)

# iteration
bellman_single_arg(w) = bellman_operator(w, m.grid, m.β, m.u,
coleman_single_arg(g) = coleman_operator(g, m.grid, m.β, m.u_prime,
m.f, m.f_prime, shocks)

# iteration
bellman_single_arg(w) = bellman_operator(w, m.grid, m.β, m.u,
coleman_single_arg(g) = coleman_operator(g, m.grid, m.β, m.u_prime,
m.f, m.f_prime, shocks)

# iteration
bellman_single_arg(w) = bellman_operator(w, m.grid, m.β, m.u,
coleman_single_arg(g) = coleman_operator(g, m.grid, m.β, m.u_prime,
m.f, m.f_prime, shocks)

# iteration
bellman_single_arg(w) = bellman_operator(w, m.grid, m.β, m.u,
coleman_single_arg(g) = coleman_operator(g, m.grid, m.β, m.u_prime,
m.f, m.f_prime, shocks)

# iteration
bellman_single_arg(w) = bellman_operator(w, m.grid, m.β, m.u,
coleman_single_arg(g) = coleman_operator(g, m.grid, m.β, m.u_prime,
m.f, m.f_prime, shocks)

# iteration
bellman_single_arg(w) = bellman_operator(w, m.grid, m.β, m.u,
coleman_single_arg(g) = coleman_operator(g, m.grid, m.β, m.u_prime,
m.f, m.f_prime, shocks)

# iteration
bellman_single_arg(w) = bellman_operator(w, m.grid, m.β, m.u,
coleman_single_arg(g) = coleman_operator(g, m.grid, m.β, m.u_prime,
m.f, m.f_prime, shocks)

# iteration
bellman_single_arg(w) = bellman_operator(w, m.grid, m.β, m.u,
coleman_single_arg(g) = coleman_operator(g, m.grid, m.β, m.u_prime,
m.f, m.f_prime, shocks)

# iteration
bellman_single_arg(w) = bellman_operator(w, m.grid, m.β, m.u,
coleman_single_arg(g) = coleman_operator(g, m.grid, m.β, m.u_prime,
m.f, m.f_prime, shocks)

# iteration
bellman_single_arg(w) = bellman_operator(w, m.grid, m.β, m.u,
coleman_single_arg(g) = coleman_operator(g, m.grid, m.β, m.u_prime,
fig, ax = subplots()
ax[plot](m.grid, g, lw=2, alpha=0.6, label="policy iteration")
ax[plot](m.grid, vf_g, lw=2, alpha=0.6, label="value iteration")
ax[legend](loc="upper left")
```

exercise2(m_ex, shocks, m.grid, m.u(m.grid), sim_length=20)

Heres the resulting figure
The policies are indeed close

**Solution to Exercise 4**

Here's the code

It assumes that you've just run the code from the previous exercise

```julia

function exercise3(m::Model, shocks::AbstractVector)
    bellman_single_arg(w) = bellman_operator(w, m.grid, m.β, m.u,
                                             m.f, shocks)
    coleman_single_arg(g) = coleman_operator(g, m.grid, m.β, m.u_prime,
                                             m.f, m.f_prime, shocks)

```

3.9. Optimal Growth II: Time Iteration
```julia
println("Timing value function iteration")
@time iterate_updating(bellman_single_arg, m.u.(m.grid), sim_length=20)
println("Timing Coleman policy function iteration")
@time iterate_updating(coleman_single_arg, m.grid, sim_length=20)
return nothing
end
exerci33(m_ex, shocks)

Timing value function iteration
  0.688099 seconds (4.08 M allocations: 367.814 MB, 4.53% gc time)
Timing Coleman policy function iteration
  20.729765 seconds (256.97 M allocations: 6.887 GB, 2.37% gc time)
```

### 3.10 Optimal Growth III: The Endogenous Grid Method

#### Contents

- *Optimal Growth III: The Endogenous Grid Method*
  - Overview
  - Key Idea
  - Implementation
  - Speed

#### 3.10.1 Overview

We solved the stochastic optimal growth model using
1. value function iteration
2. Euler equation based time iteration

We found time iteration to be significantly more accurate at each step

In this lecture we look at an ingenious twist on the time iteration technique called the endogenous grid method (EGM)

EGM is a numerical method for implementing policy iteration invented by Chris Carroll

It is a good example of how a clever algorithm can save a massive amount of computer time

(Massive when we multiply saved CPU cycles on each implementation times the number of implementations worldwide)
The original reference is [Car06]

3.10.2 Key Idea

Let's start by reminding ourselves of the theory and then see how the numerics fit in

Theory

Take the model set out in the time iteration lecture, following the same terminology and notation.

The Euler equation is

\[
(u' \circ c^*)(y) = \beta \int (u' \circ c^*)(f(y - c^*(y))z)f'(y - c^*(y))z\phi(dz)
\] (3.56)

As we saw, the Coleman operator is a nonlinear operator $K$ engineered so that $c^*$ is a fixed point of $K$.

It takes as its argument a continuous strictly increasing consumption policy $g \in \Sigma$.

It returns a new function $Kg$, where $(Kg)(y)$ is the $c \in (0, \infty)$ that solves

\[
u'(c) = \beta \int (u' \circ g)(f(y - c)z)f'(y - c)z\phi(dz)
\] (3.57)

Exogenous Grid

As discussed in the lecture on time iteration, to implement the method on a computer we need numerical approximation.

In particular, we represent a policy function by a set of values on a finite grid.

The function itself is reconstructed from this representation when necessary, using interpolation or some other method.

Previously, to obtain a finite representation of an updated consumption policy we

- fixed a grid of income points \( \{y_i\} \)
- calculated the consumption value $c_i$ corresponding to each $y_i$ using (3.57) and a root finding routine.

Each $c_i$ is then interpreted as the value of the function $Kg$ at $y_i$.

Thus, with the points \( \{y_i, c_i\} \) in hand, we can reconstruct $Kg$ via approximation.

Iteration then continues.
Endogenous Grid

The method discussed above requires a root finding routine to find the $c_i$ corresponding to a given income value $y_i$.

Root finding is costly because it typically involves a significant number of function evaluations. As pointed out by Carroll [Car06], we can avoid this if $y_i$ is chosen endogenously.

The only assumption required is that $u'$ is invertible on $(0, \infty)$.

The idea is this:
First we fix an exogenous grid $\{k_i\}$ for capital ($k = y - c$).
Then we obtain $c_i$ via
\[
c_i = (u')^{-1} \left\{ \beta \int (u' \circ g)(f(k_i)z) f'(k_i) z \phi(dz) \right\}
\]  
(3.58)

where $(u')^{-1}$ is the inverse function of $u'$.

Finally, for each $c_i$ we set $y_i = c_i + k_i$.

It is clear that each $(y_i, c_i)$ pair constructed in this manner satisfies (3.57).

With the points $\{y_i, c_i\}$ in hand, we can reconstruct $Kg$ via approximation as before.

The name EGM comes from the fact that the grid $\{y_i\}$ is determined endogenously.

3.10.3 Implementation

Let's implement this version of the Coleman operator and see how it performs.

The Operator

Here's an implementation of $K$ using EGM as described above:

```julia
#=
Authors: Shunsuke Hori
=

The approximate Coleman operator, updated using the endogenous grid method.

Parameters
----------
g : Function
    The current guess of the policy function
```
function coleman_egm(g::Function, k_grid::AbstractVector, β::AbstractFloat, u_prime::Function, u_prime_inv::Function, f::Function, f_prime::Function, shocks::AbstractVector)

# Allocate memory for value of consumption on endogenous grid points
    c = similar(k_grid)

# Solve for updated consumption value
for (i, k) in enumerate(k_grid)
    vals = u_prime.(g.(f(k) * shocks)) .* f_prime(k) .* shocks
    c[i] = u_prime_inv(β * mean(vals))
end

# Determine endogenous grid
    y = k_grid + c  # y_i = k_i + c_i

# Update policy function and return
    Kg = LinInterp(y, c)
    Kg_f(x) = Kg(x)
    return Kg_f
end

Note the lack of any root finding algorithm

Well also run our original implementation, which uses an exogenous grid and requires root finding, so we can perform some comparisons

#=

Author: Shunsuke Hori
```julia
using QuantEcon

"""
g: input policy function
grid: grid points
\( \beta \): discount factor
u_prime: derivative of utility function
f: production function
f_prime: derivative of production function
shocks::shock draws, used for Monte Carlo integration to compute expectation
Kg: output value is stored
"""

function coleman_operator!(g::AbstractVector,
grid::AbstractVector,
\( \beta \)::AbstractFloat,
u_prime::Function,
f::Function,
f_prime::Function,
shocks::AbstractVector,
Kg::AbstractVector=similar(g))

    # This function requires the container of the output value as argument Kg

    # Construct linear interpolation object#
g_func = LinInterp(grid, g)

    # solve for updated consumption value#
    for (i, y) in enumerate(grid)
        function h(c)
            vals = u_prime.(g_func.(f(y - c) + shocks)) .* f_prime(y - c) .* shocks
            return u_prime(c) - \( \beta \) * mean(vals)
        end
        Kg[i] = brent(h, 1e-10, y - 1e-10)
    end
    return Kg
end

# The following function does NOT require the container of the output value as argument

function coleman_operator(g::AbstractVector,
grid::AbstractVector,
\( \beta \)::AbstractFloat,
u_prime::Function,
f::Function,
f_prime::Function,
shocks::AbstractVector)

    return coleman_operator!(g, grid, \( \beta \), u_prime,
f, f_prime, shocks, similar(g))
end
```

Chapter 3. Dynamic Programming
Lets test out the code above on some example parameterizations, after the following imports

```julia
using PyPlot
using LaTeXStrings
```

### Testing on the Log / Cobb–Douglas case

As we did for value function iteration and time iteration, lets start by testing our method with the log-linear benchmark

The first step is to bring in the model that we used in the Coleman policy function iteration

```julia
#= 
Author: Shunsuke Hori
=#

struct Model{TF <: AbstractFloat, TR <: Real, TI <: Integer}
    α::TR    # Productivity parameter
    β::TF    # Discount factor
    γ::TR    # risk aversion
    μ::TR    # First parameter in lognorm(μ, σ)
    s::TR    # Second parameter in lognorm(μ, σ)
    grid_min::TR # Smallest grid point
    grid_max::TR # Largest grid point
    grid_size::TI # Number of grid points
    u::Function # utility function
    u_prime::Function # derivative of utility function
    f::Function # production function
    f_prime::Function # derivative of production function
    grid::Vector{TR} # grid
end
```

### construct Model instance using the information of parameters and functional form

arguments: see above

return: Model type instance

```julia
function Model(; α::Real=0.65, # Productivity parameter
    β::AbstractFloat=0.95, # Discount factor
    γ::Real=1.0, # risk aversion
    μ::Real=0.0, # First parameter in lognorm(μ, σ)
    s::Real=0.1, # Second parameter in lognorm(μ, σ)
    grid_min::Real=1e-6, # Smallest grid point
    grid_max::Real=4.0, # Largest grid point
    grid_size::Integer=200, # Number of grid points
    )
```

---

3.10. Optimal Growth III: The Endogenous Grid Method 445
u::Function = c->(c^((1-\gamma)-1))/(1-\gamma), # utility function
u_prime::Function = c-> c^(-\gamma), # u'
f::Function = k-> k^\alpha, # production function
f_prime::Function = k -> \alpha + k^{(\alpha-1)} # f'

grid = collect(linspace(grid_min, grid_max, grid_size))

if \gamma == 1 # when \gamma=1, log utility
    u_log(c) = log(c)
    m = Model(\alpha, \beta, \gamma, \mu, s, grid_min, grid_max,
              grid_size, u_log, u_prime, f, f_prime, grid)
else
    m = Model(\alpha, \beta, \gamma, \mu, s, grid_min, grid_max,
              grid_size, u, u_prime, f, f_prime, grid)
end
return m
end

Next we generate an instance

mlog = Model(\gamma=1.0) # Log Linear model

We also need some shock draws for Monte Carlo integration

shock_size = 250 # Number of shock draws in Monte Carlo integral
shocks = exp(mlog.\mu + mlog.s * randn(shock_size))

As a preliminary test, lets see if Kc^* = c^*, as implied by the theory

c_star(y) = (1 - mlog.\alpha * mlog.\beta) * y

# == Some useful constants == #
ab = mlog.\alpha * mlog.\beta

\nc1 = log(1 - ab) / (1 - mlog.\beta)
\nc2 = (mlog.\mu + mlog.\alpha * log(ab)) / (1 - mlog.\alpha)
\nc3 = 1 / (1 - mlog.\beta)
\nc4 = 1 / (1 - ab)

v_star(y) = c1 + c2 * (c3 - c4) + c4 * log(y)

function verify_true_policy(m::Model,
                            shocks::AbstractVector,
                            c_star::Function)
    k_grid = m.grid
    c_star_new = coleman_egm(c_star, k_grid, m.\beta, m.u_prime,
                            m.u_prime, m.\alpha, m.f, m.f_prime, shocks)
    fig, ax = subplots(figsize=(9, 6))

    ax[:plot](k_grid, c_star.(k_grid), label="Optimal policy $c^*$")
    ax[:plot](k_grid, c_star_new.(k_grid), label="$Kc^*$")
Notice that we are passing \( u' \) to \( coleman_egm \) twice. The reason is that, in the case of log utility, \( u'(c) = (u')^{-1}(c) = 1/c \).

Hence \( u' \) and \( u'_i \) are the same.

In any case, here is the result:

![Graph showing optimal policy \( c^* \) and \( Kc^* \)](image)

We can't really distinguish the two plots.

In fact it is easy to see that the difference is essentially zero:

```plaintext
c_star_new = coleman_egm(c_star, mlog.grid, mlog.β, mlog.u_prime, mlog.u_prime, mlog.f, mlog.f_prime, shocks)
maximum(abs, (c_star_new.(mlog.grid) - c_star.(mlog.grid)))
```

```
9.881666666666682e-6
```

Next let's try iterating from an arbitrary initial condition and see if we converge towards \( c^* \).

Let's start from the consumption policy that eats the whole pie: \( c(y) = y \)
g_init(x) = x
n = 15

```julia
function check_convergence(m::Model,
    shocks::AbstractVector,
    c_star::Function,
    g_init::Function,
    n_iter::Integer)

    k_grid = m.grid
    g = g_init
    fig, ax = subplots(figsize = (9, 6))
    jet = ColorMap("jet")
    plot(m.grid, g.(m.grid),
        color=jet(0), lw=2, alpha=0.6, label="initial condition $c(y) = y$")
    for i in 1:n_iter
        new_g = coleman_egm(g, k_grid, 
            m.β, m.u_prime, m.u_prime, m.f, m.f_prime, shocks)
        g = new_g
        ax[:plot](k_grid, new_g.(k_grid), alpha=0.6, color=jet(i / n_iter), lw=2)
    end
    ax[:plot](k_grid, c_star.(k_grid), 
        "k-", lw=2, alpha=0.8, label="true policy function $c^*$")
    ax[:legend](loc="upper left")
end

check_convergence(mlog, shocks, c_star, g_init, n)
```

We see that the policy has converged nicely, in only a few steps
3.10.4 Speed

Now let’s compare the clock times per iteration for the standard Coleman operator (with exogenous grid) and the EGM version.

We’ll do so using the CRRA model adopted in the exercises of the *Euler equation time iteration lecture*.

Here’s the model and some convenient functions:

```julia
mcrra = Model(α = 0.65, β = 0.95, γ = 1.5)
u_prime_inv(c) = c^(-1/mcrra.γ)
```

Here’s the result:

```julia
function compare_clock_time(m::Model,
                           u_prime_inv::Function;
                           sim_length::Integer=20)
    k_grid = m.grid
crra_coleman(g) = coleman_operator(g, k_grid, m.β, m.u_prime,
                                     m.f, m.f_prime, shocks)
crra_coleman_egm(g) = coleman_egm(g, k_grid, m.β, m.u_prime,
                                     u_prime_inv, m.f, m.f_prime, shocks)
    print("Timing standard Coleman policy function iteration")
g_init = k_grid
```

3.10. Optimal Growth III: The Endogenous Grid Method
```julia
    g = g_init
    @time begin
        for i in 1:sim_length
            new_g = crra_coleman(g)
            g = new_g
        end
    end

    print("Timing policy function iteration with endogenous grid")
    g_init_egm(x) = x
    g = g_init_egm
    @time begin
        for i in 1:sim_length
            new_g = crra_coleman_egm(g)
            g = new_g
        end
    end
    return nothing
end
```

Timing standard Coleman policy function iteration 7.314275 seconds (53.48 M, ~allocations: 1.227 GB, 1.77% gc time)
Timing policy function iteration with endogenous grid 0.776051 seconds (5.24 M allocations: 116.309 MB, 1.39% gc time)

We see that the EGM version is more than 9 times faster

At the same time, the absence of numerical root finding means that it is typically more accurate at each step as well

### 3.11 LQ Dynamic Programming Problems

**Contents**

- *LQ Dynamic Programming Problems*
  - Overview
  - Introduction
  - Optimality – Finite Horizon
  - Implementation
  - Extensions and Comments
  - Further Applications
  - Exercises
3.11.1 Overview

Linear quadratic (LQ) control refers to a class of dynamic optimization problems that have found applications in almost every scientific field.

This lecture provides an introduction to LQ control and its economic applications.

As we will see, LQ systems have a simple structure that makes them an excellent workhorse for a wide variety of economic problems.

Moreover, while the linear-quadratic structure is restrictive, it is in fact far more flexible than it may appear initially.

These themes appear repeatedly below.

Mathematically, LQ control problems are closely related to the Kalman filter.

- Recursive formulations of linear-quadratic control problems and Kalman filtering problems both involve matrix **Riccati equations**.
- Classical formulations of linear control and linear filtering problems make use of similar matrix decompositions (see for example these lectures).

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 these lectures).

For additional reading on LQ control, see, for example,

- [LS18], chapter 5
- [HS08], chapter 4
- [HLL96], section 3.5

In order to focus on computation, we leave longer proofs to these sources (while trying to provide as much intuition as possible).

3.11.2 Introduction

The linear part of LQ is a linear law of motion for the state, while the quadratic part refers to preferences.

Let’s begin with the former, move on to the latter, and then put them together into an optimization problem.
The Law of Motion

Let $x_t$ be a vector describing the state of some economic system.

Suppose that $x_t$ follows a linear law of motion given by

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \quad t = 0, 1, 2, \ldots$$

(3.59)

Here

- $u_t$ is a control vector, incorporating choices available to a decision maker confronting the current state $x_t$.
- $\{w_t\}$ is an uncorrelated zero mean shock process satisfying $\mathbb{E}w_tw'_t = I$, where the right-hand side is the identity matrix.

Regarding the dimensions:

- $x_t$ is $n \times 1$, $A$ is $n \times n$.
- $u_t$ is $k \times 1$, $B$ is $n \times k$.
- $w_t$ is $j \times 1$, $C$ is $n \times j$.

Example 1

Consider a household budget constraint given by

$$a_{t+1} + c_t = (1 + r)a_t + y_t$$

Here $a_t$ is assets, $r$ is a fixed interest rate, $c_t$ is current consumption, and $y_t$ is current non-financial income.

If we suppose that $\{y_t\}$ is serially uncorrelated and $N(0, \sigma^2)$, then, taking $\{w_t\}$ to be standard normal, we can write the system as

$$a_{t+1} = (1 + r)a_t - c_t + \sigma w_{t+1}$$

This is clearly a special case of (3.59), with assets being the state and consumption being the control.

Example 2

One unrealistic feature of the previous model is that non-financial income has a zero mean and is often negative.

This can easily be overcome by adding a sufficiently large mean.

Hence in this example we take $y_t = \sigma w_{t+1} + \mu$ for some positive real number $\mu$.

Another alteration that's useful to introduce (well see why soon) is to change the control variable from consumption to the deviation of consumption from some ideal quantity $\bar{c}$.
(Most parameterizations will be such that $\bar{c}$ is large relative to the amount of consumption that is attainable in each period, and hence the household wants to increase consumption)

For this reason, we now take our control to be $u_t := c_t - \bar{c}$

In terms of these variables, the budget constraint $a_{t+1} = (1 + r)a_t - c_t + y_t$ becomes

$$a_{t+1} = (1 + r)a_t - u_t - \bar{c} + \sigma w_{t+1} + \mu$$  \hspace{1cm} (3.60)

How can we write this new system in the form of equation (3.59)?

If, as in the previous example, we take $a_t$ as the state, then we run into a problem: the law of motion contains some constant terms on the right-hand side.

This means that we are dealing with an affine function, not a linear one (recall this discussion).

Fortunately, we can easily circumvent this problem by adding an extra state variable.

In particular, if we write

$$
\begin{pmatrix}
a_{t+1} \\
1
\end{pmatrix}
= 
\begin{pmatrix}
1 + r & -\bar{c} + \mu \\
0 & 1
\end{pmatrix}
\begin{pmatrix}
a_t \\
1
\end{pmatrix}
+ 
\begin{pmatrix}
-1 \\
0
\end{pmatrix}
u_t
+ 
\begin{pmatrix}
\sigma \\
0
\end{pmatrix}w_{t+1}
$$  \hspace{1cm} (3.61)

then the first row is equivalent to (3.60)

Moreover, the model is now linear, and can be written in the form of (3.59) by setting

$$x_t := \begin{pmatrix} a_t \\ 1 \end{pmatrix}, \quad A := \begin{pmatrix} 1 + r & -\bar{c} + \mu \\ 0 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \end{pmatrix}$$  \hspace{1cm} (3.62)

In effect, we've bought ourselves linearity by adding another state.

**Preferences**

In the LQ model, the aim is to minimize a flow of losses, where time-$t$ loss is given by the quadratic expression

$$x_t'Rx_t + u_t'Qu_t$$  \hspace{1cm} (3.63)

Here

- $R$ is assumed to be $n \times n$, symmetric and nonnegative definite
- $Q$ is assumed to be $k \times k$, symmetric and positive definite

**Note:** In fact, for many economic problems, the definiteness conditions on $R$ and $Q$ can be relaxed. It is sufficient that certain submatrices of $R$ and $Q$ be nonnegative definite. See [HS08] for details.
Example 1

A very simple example that satisfies these assumptions is to take \( R \) and \( Q \) to be identity matrices, so that current loss is

\[
x_t' I x_t + u_t' I u_t = \|x_t\|^2 + \|u_t\|^2
\]

Thus, for both the state and the control, loss is measured as squared distance from the origin.

(In fact the general case (3.63) can also be understood in this way, but with \( R \) and \( Q \) identifying other – non-Euclidean – notions of distance from the zero vector)

Intuitively, we can often think of the state \( x_t \) as representing deviation from a target, such as

- deviation of inflation from some target level
- deviation of a firm’s capital stock from some desired quantity

The aim is to put the state close to the target, while using controls parsimoniously

Example 2

In the household problem studied above, setting \( R = 0 \) and \( Q = 1 \) yields preferences

\[
x_t' R x_t + u_t' Q u_t = u_t^2 = (c_t - \bar{c})^2
\]

Under this specification, the household’s current loss is the squared deviation of consumption from the ideal level \( \bar{c} \)

3.11.3 Optimality – Finite Horizon

Let’s now be precise about the optimization problem we wish to consider, and look at how to solve it

The Objective

We will begin with the finite horizon case, with terminal time \( T \in \mathbb{N} \)

In this case, the aim is to choose a sequence of controls \( \{u_0, \ldots, u_{T-1}\} \) to minimize the objective

\[
\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (x_t' R x_t + u_t' Q u_t) + \beta^T x_T' R_f x_T \right\} \tag{3.64}
\]

subject to the law of motion (3.59) and initial state \( x_0 \)

The new objects introduced here are \( \beta \) and the matrix \( R_f \)

The scalar \( \beta \) is the discount factor, while \( x' R_f x \) gives terminal loss associated with state \( x \)

Comments:
We assume $R_f$ to be $n \times n$, symmetric and nonnegative definite.

- We allow $\beta = 1$, and hence include the undiscounted case.
- $x_0$ may itself be random, in which case we require it to be independent of the shock sequence $w_1, \ldots, w_T$.

### Information

There's one constraint we've neglected to mention so far, which is that the decision maker who solves this LQ problem knows only the present and the past, not the future.

To clarify this point, consider the sequence of controls $\{u_0, \ldots, u_{T-1}\}$.

When choosing these controls, the decision maker is permitted to take into account the effects of the shocks $\{w_1, \ldots, w_T\}$ on the system.

However, it is typically assumed and will be assumed here that the time-$t$ control $u_t$ can be made with knowledge of past and present shocks only.

The fancy measure-theoretic way of saying this is that $u_t$ must be measurable with respect to the $\sigma$-algebra generated by $x_0, w_1, w_2, \ldots, w_t$.

This is in fact equivalent to stating that $u_t$ can be written in the form $u_t = g_t(x_0, w_1, w_2, \ldots, w_t)$ for some Borel measurable function $g_t$.

(Just about every function that's useful for applications is Borel measurable, so, for the purposes of intuition, you can read that last phrase as for some function $g_t$.)

Now note that $x_t$ will ultimately depend on the realizations of $x_0, w_1, w_2, \ldots, w_t$.

In fact it turns out that $x_t$ summarizes all the information about these historical shocks that the decision maker needs to set controls optimally.

More precisely, it can be shown that any optimal control $u_t$ can always be written as a function of the current state alone.

Hence in what follows we restrict attention to control policies (i.e., functions) of the form $u_t = g_t(x_t)$.

Actually, the preceding discussion applies to all standard dynamic programming problems.

What's special about the LQ case is that -- as we shall soon see -- the optimal $u_t$ turns out to be a linear function of $x_t$.

### Solution

To solve the finite horizon LQ problem we can use a dynamic programming strategy based on backwards induction that is conceptually similar to the approach adopted in this lecture.

For reasons that will soon become clear, we first introduce the notation $J_T(x) = x' R_f x$.

Now consider the problem of the decision maker in the second to last period.

In particular, let the time be $T - 1$, and suppose that the state is $x_{T-1}$.
The decision maker must trade off current and (discounted) final losses, and hence solves
\[
\min_u \{x_{T-1}'Rx_{T-1} + u'Qu + \beta EJ_T(Ax_{T-1} + Bu + Cw_T)\}
\]
At this stage, it is convenient to define the function
\[
J_{T-1}(x) = \min_u \{x'R + u'Qu + \beta EJ_T(Ax + Bu + Cw_T)\}
\] (3.65)
The function \(J_{T-1}\) will be called the \(T-1\) value function, and \(J_{T-1}(x)\) can be thought of as representing total loss-to-go from state \(x\) at time \(T-1\) when the decision maker behaves optimally.

Now lets 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
\[
\min_u \{x_{T-2}'Rx_{T-2} + u'Qu + \beta EJ_{T-1}(Ax_{T-2} + Bu + Cw_{T-1})\}
\]
Letting
\[
J_{T-2}(x) = \min_u \{x'R + u'Qu + \beta EJ_{T-1}(Ax + Bu + Cw_{T-1})\}
\]
the pattern for backwards induction is now clear.

In particular, we define a sequence of value functions \(\{J_0, \ldots, J_T\}\) via
\[
J_t(x) = \min_u \{x'R + u'Qu + \beta EJ_{t+1}(Ax + Bu + Cw_t)\}, \quad \text{and} \quad J_T(x) = x'R_f x
\]
The first equality is the Bellman equation from dynamic programming theory specialized to the finite horizon LQ problem.

Now that we have \(\{J_0, \ldots, J_T\}\), we can obtain the optimal controls.

As a first step, lets 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 an \(n \times n\) 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, (3.65) becomes
\[
J_{T-1}(x) = \min_u \{x'R + u'Qu + \beta E(Ax + Bu + Cw_T)'P_T(Ax + Bu + Cw_T)\}
\] (3.66)
To obtain the minimizer, we can take the derivative of the r.h.s. with respect to $u$ and set it equal to zero.

Applying the relevant rules of *matrix calculus*, this gives

$$
  u = -(Q + \beta B'P_T B)^{-1}\beta B'P_T Ax
$$

(3.67)

Plugging this back into (3.66) and rearranging yields

$$
  J_{T-1}(x) = x'P_{T-1}x + d_{T-1}
$$

where

$$
  P_{T-1} = R - \beta^2 A'P_T B(Q + \beta B'P_T B)^{-1}B'P_T A + \beta A'P_T A
$$

(3.68)

and

$$
  d_{T-1} := \beta \text{trace}(C'P_T C)
$$

(3.69)

(The algebra is a good exercise well leave it up to you)

If we continue working backwards in this manner, it soon becomes clear that $J_t(x) = x'P_t x + d_t$ as claimed, where $\{P_t\}$ and $\{d_t\}$ satisfy the recursions

$$
  P_{t-1} = R - \beta^2 A'P_t B(Q + \beta B'P_t B)^{-1}B'P_t A + \beta A'P_t A \quad \text{with} \quad P_T = R_f
$$

(3.70)

and

$$
  d_{t-1} = \beta(d_t + \text{trace}(C'P_t C)) \quad \text{with} \quad d_T = 0
$$

(3.71)

Recalling (3.67), the minimizers from these backward steps are

$$
  u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B'P_{t+1} B)^{-1}\beta B'P_{t+1} A
$$

(3.72)

These are the linear optimal control policies we *discussed above*

In particular, the sequence of controls given by (3.72) and (3.59) solves our finite horizon LQ problem.

Rephrasing this more precisely, the sequence $u_0, \ldots, u_{T-1}$ given by

$$
  u_t = -F_t x_t \quad \text{with} \quad x_{t+1} = (A - BF_t)x_t + Cw_{t+1}
$$

(3.73)

for $t = 0, \ldots, T - 1$ attains the minimum of (3.64) subject to our constraints
3.11.4 Implementation

We will use code from lqcontrol.jl in QuantEcon.jl to solve finite and infinite horizon linear quadratic control problems.

In the module, the various updating, simulation and fixed point methods act on 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 $C = None$ (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 (3.70), (3.71) and (3.72)
  - `stationary_values` computes $P, d, F$ in the infinite horizon case
  - `compute_sequence` - simulates the dynamics of $x_t, u_t, w_t$ given $x_0$ and assuming standard normal shocks

An Application

Early Keynesian models assumed that households have a constant marginal propensity to consume from current income.

Data contradicted the constancy of the marginal propensity to consume.

In response, Milton Friedman, Franco Modigliani and others built models based on a consumers preference for an intertemporally smooth consumption stream.

(See, for example, [Fri56] or [MB54])

One property of those models is that households purchase and sell financial assets to make consumption streams smoother than income streams.

The household savings problem **outlined above** captures these ideas.

The optimization problem for the household is to choose a consumption sequence in order to minimize

$$
E \left\{ \sum_{t=0}^{T-1} \beta^t (c_t - \bar{c})^2 + \beta^T qa_T^2 \right\}
$$

subject to the sequence of budget constraints $a_{t+1} = (1 + r)a_t - c_t + y_t$, $t \geq 0$. 

458 Chapter 3. Dynamic Programming
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 (3.60)

We saw how this constraint could be manipulated into the LQ formulation $x_{t+1} = Ax_t + Bu_t + Cw_{t+1}$ by setting $x_t = (a_t, 1)'$ and using the definitions in (3.62)

To match with this state and control, the objective function (3.74) can be written in the form of (3.64) by choosing

$$Q := 1, \quad R := \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad \text{and} \quad R_f := \begin{pmatrix} q & 0 \\ 0 & 0 \end{pmatrix}$$

Now that the problem is expressed in LQ form, we can proceed to the solution by applying (3.70) and (3.72)

After generating shocks $w_1, \ldots, w_T$, the dynamics for assets and consumption can be simulated via (3.73)

The following figure was computed using $r = 0.05, \beta = 1/(1 + r), \bar{c} = 2, \mu = 1, \sigma = 0.25, T = 45$ and $q = 10^6$

The shocks $\{w_t\}$ were taken to be iid and standard normal

```julia
using QuantEcon
using Plots
using Plots.PlotMeasures
pyplot()

# == Model parameters == #
r = 0.05
beta = 1/(1 + r)
T = 45
c_bar = 2.0
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; 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=beta, capT=T, rf=Rf)
x0 = [0.0; 1.0]
xp, up, wp = compute_sequence(lq, x0)

# == Convert back to assets, consumption and income ==#
```

3.11. LQ Dynamic Programming Problems
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 re-
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.

```julia
# == Compute solutions and simulate == #
lq = LQ(Q, R, A, B, C; bet=0.96, capT=T, rf=Rf)
x0 = [0.0; 1.0]
xp, up, wp = compute_sequence(lq, x0)

# == Convert back to assets, consumption and income == #
assets = vec(xp[1, :])  # a_t
  c = vec(up + c_bar)  # c_t
income = vec(\sigma * wp[1, 2:end] + \mu)  # y_t

# == Plot results == #
p = plot(Vector[assets, c, zeros(T + 1), income, cumsum(income - \mu)],
    lab=["assets" "consumption" "non-financial income" "cumulative unanticipated income"],
    color=[:blue :green :black :orange :red],
    xaxis="Time", layout=(2, 1),
    bottom_margin=20mm, size=(600, 600))
```
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.

### 3.11.5 Extensions and Comments

Let’s now consider a number of standard extensions to the LQ problem treated above.

**Time-Varying Parameters**

In some settings it can be desirable to allow $A, B, C, R$ and $Q$ to depend on $t$.

For the sake of simplicity, we’ve chosen not to treat this extension in our implementation given below.
However, the loss of generality is not as large as you might first imagine. In fact, we can tackle many models with time-varying parameters by suitable choice of state variables. One illustration is given below.

For further examples and a more systematic treatment, see [HS13], section 2.4.

**Adding a Cross-Product Term**

In some LQ problems, preferences include a cross-product term $u_t'N x_t$, so that the objective function becomes

$$\mathbb{E}\left\{ \sum_{t=0}^{T-1} \beta^t(x_t'Rx_t + u_t'Qu_t + 2u_t'N x_t) + \beta^T x_T'R_f x_T \right\} \quad (3.75)$$

Our results extend to this case in a straightforward way. The sequence $\{P_t\}$ from (3.70) becomes

$$P_{t-1} = R - (\beta B' P_t A + N)'(Q + \beta B' P_t B)^{-1}(\beta B' P_t A + N) + \beta A' P_t A \quad \text{with} \quad P_T = R_f \quad (3.76)$$

The policies in (3.72) are modified to

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B' P_{t+1} B)^{-1}(\beta B' P_{t+1} A + N) \quad (3.77)$$

The sequence $\{d_t\}$ is unchanged from (3.71).

We leave interested readers to confirm these results (the calculations are long but not overly difficult).

**Infinite Horizon**

Finally, we consider the infinite horizon case, with *cross-product term*, unchanged dynamics and objective function given by

$$\mathbb{E}\left\{ \sum_{t=0}^{\infty} \beta^t(x_t'Rx_t + u_t'Qu_t + 2u_t'N x_t) \right\} \quad (3.78)$$

In the infinite horizon case, optimal policies can depend on time only if time itself is a component of the state vector $x_t$.

In other words, there exists a fixed matrix $F$ such that $u_t = -Fx_t$ for all $t$.

That decision rules are constant over time is intuitive after all, the decision maker faces the same infinite horizon at every stage, with only the current state changing.
Not surprisingly, $P$ and $d$ are also constant.

The stationary matrix $P$ is the solution to the discrete time algebraic Riccati equation

$$P = R - (\beta B'PA + N)'(Q + \beta B'PB)^{-1}(\beta B'PA + N) + \beta A'PA \quad (3.79)$$

Equation (3.79) is also called the LQ Bellman equation, and the map that sends a given $P$ into the right-hand side of (3.79) is called the LQ Bellman operator.

The stationary optimal policy for this model is

$$u = -Fx \quad \text{where} \quad F = (Q + \beta B'PB)^{-1}(\beta B'PA + N) \quad (3.80)$$

The sequence $\{d_t\}$ from (3.71) is replaced by the constant value

$$d := \text{trace}(C'PC) \frac{\beta}{1-\beta} \quad (3.81)$$

The state evolves according to the time-homogeneous process $x_{t+1} = (A - BF)x_t + Cw_{t+1}$

An example infinite horizon problem is treated below.

**Certainty Equivalence**

Linear quadratic control problems of the class discussed above have the property of certainty equivalence. By this we mean that the optimal policy $F$ is not affected by the parameters in $C$, which specify the shock process. This can be confirmed by inspecting (3.80) or (3.77).

It follows that we can ignore uncertainty when solving for optimal behavior, and plug it back in when examining optimal state dynamics.

### 3.11.6 Further Applications

**Application 1: Age-Dependent Income Process**

Previously we studied a permanent income model that generated consumption smoothing.

One unrealistic feature of that model is the assumption that the mean of the random income process does not depend on the consumers 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

$$\mathbb{E}\left\{ \sum_{t=0}^{T-1} \beta^t (c_t - \bar{c})^2 + \beta^T qa_T^2 \right\}$$

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

$$a_{t+1} = (1 + r)a_t - u_t - \bar{c} + m_1 t + m_2 t^2 + \sigma w_{t+1}$$

(3.83)

The fact that $a_{t+1}$ is a linear function of $(a_t, 1, t, t^2)$ suggests taking these four variables as the state vector $x_t$

Once a good choice of state and control (recall $u_t = c_t - \bar{c}$) has been made, the remaining specifications fall into place relatively easily

Thus, for the dynamics we set

$$x_t := \begin{pmatrix} a_t \\ 1 \\ t \\ t^2 \end{pmatrix}, \quad A := \begin{pmatrix} 1 + r & -\bar{c} & m_1 & m_2 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

(3.84)

If you expand the expression $x_{t+1} = Ax_t + Bu_t + Cw_{t+1}$ using this specification, you will find that assets follow (3.83) as desired, and that the other state variables also update appropriately

To implement preference specification (3.82) we take

$$Q := 1, \quad R := \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ and } R_f := \begin{pmatrix} q & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

(3.85)

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

$$y_t = \begin{cases} 
p(t) + \sigma w_{t+1} & \text{if } t \leq K \\
\ s & \text{otherwise} \end{cases}$$  \hspace{1cm} (3.86)

Here

- $p(t) := m_1 t + m_2 t^2$ with the coefficients $m_1, m_2$ chosen such that $p(K) = \mu$ and $p(0) = p(2K) = 0$
- $s$ is retirement income

We suppose that preferences are unchanged and given by (3.74)

The budget constraint is also unchanged and given by $a_{t+1} = (1 + r) a_t - c_t + y_t$

Our aim is to solve this problem and simulate paths using the LQ techniques described in this lecture

In fact this is a nontrivial problem, as the kink in the dynamics (3.86) at $K$ makes it very difficult to express the law of motion as a fixed-coefficient linear system

However, we can still use our LQ methods here by suitably linking two component LQ problems

These two LQ problems describe the consumers behavior during her working life ($lq\text{-}working$) and retirement ($lq\text{-}retired$)

(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 $lq\text{-}retired$ 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 $lq\text{-}working$ specification
3. solve $lq\text{-}working$ 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

\[
E \left\{ \sum_{t=0}^{\infty} \beta^t \pi_t \right\} \quad \text{where} \quad \pi_t := p_t q_t - c q_t - \gamma (q_{t+1} - q_t)^2
\]  

(3.87)

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 \( \gamma \).
dynamics with $\gamma = 1$
dynamics with $\gamma = 10$
To produce these figures we converted the monopolist problem into an LQ problem.

The key to this conversion is to choose the right state which can be a bit of an art. Here we take \( x_t = (\bar{q}_t, q_t, 1) \), while the control is chosen as \( u_t = q_{t+1} - q_t \).

We also manipulated the profit function slightly. In (3.87), current profits are \( \pi_t := p_t q_t - c q_t - \gamma (q_{t+1} - q_t)^2 \).

Let’s now replace \( \pi_t \) in (3.87) 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 (3.87), and optimizers are not affected by constant terms.)

The reason for making this substitution is that, as you will be able to verify, \( \hat{\pi}_t \) reduces to the simple quadratic

\[
\hat{\pi}_t = -a_1 (q_t - \bar{q}_t)^2 - \gamma u_t^2
\]

After negation to convert to a minimization problem, the objective becomes

\[
\min \mathbb{E} \sum_{t=0}^{\infty} \beta^t \{ a_1 (q_t - \bar{q}_t)^2 + \gamma u_t^2 \} \quad (3.88)
\]
Its now relatively straightforward to find $R$ and $Q$ such that (3.88) can be written as (3.78)

Furthermore, the matrices $A, B$ and $C$ from (3.59) can be found by writing down the dynamics of each element of the state

*Exercise 3* asks you to complete this process, and reproduce the preceding figures

### 3.11.7 Exercises

**Exercise 1**

Replicate the figure with polynomial income *shown above*

The parameters are $r = 0.05, \beta = 1/(1 + r), \bar{c} = 1.5, \mu = 2, \sigma = 0.15, T = 50$ and $q = 10^4$

**Exercise 2**

Replicate the figure on work and retirement *shown above*

The parameters are $r = 0.05, \beta = 1/(1 + r), \bar{c} = 4, \mu = 4, \sigma = 0.35, K = 40, T = 60, s = 1$ and $q = 10^4$

To understand the overall procedure, carefully read the section containing that figure

Some hints are as follows:

First, in order to make our approach work, we must ensure that both LQ problems have the same state variables and control

As with previous applications, the control can be set to $u_t = c_t - \bar{c}$

For *lq_working*, $x_t, A, B, C$ can be chosen as in (3.84)

- Recall that $m_1, m_2$ are chosen so that $p(K) = \mu$ and $p(2K) = 0$

For *lq_retired*, use the same definition of $x_t$ and $u_t$, but modify $A, B, C$ to correspond to constant income $y_t = s$

For *lq_retired*, set preferences as in (3.85)

For *lq_working*, preferences are the same, except that $R_f$ should be replaced by the final value function that emerges from iterating *lq_retired* back to the start of retirement

With some careful footwork, the simulation can be generated by patching together the simulations from these two separate models

**Exercise 3**

Reproduce the figures from the monopolist application *given above*

For parameters, use $a_0 = 5, a_1 = 0.5, \sigma = 0.15, \rho = 0.9, \beta = 0.95$ and $c = 2$, while $\gamma$ varies between 1 and 50 (see figures)
3.11.8 Solutions

Exercise 1

Here's one solution.

We use some fancy plot commands to get a certain style feel free to use simpler ones.

The model is an LQ permanent income / life-cycle model with hump-shaped income

\[ y_t = m_1 t + m_2 t^2 + \sigma w_{t+1} \]

where \( \{w_t\} \) is iid \( N(0,1) \) and the coefficients \( m_1 \) and \( m_2 \) are chosen so that \( p(t) = m_1 t + m_2 t^2 \) has an inverted U shape with

- \( p(0) = 0, p(T/2) = \mu \), and
- \( p(T) = 0 \).

```julia
# == Model parameters == #
r = 0.05
\beta = 1/(1 + r)
T = 50
c_bar = 1.5
\sigma = 0.15
\mu = 2
q = 1e4
m1 = T * (\mu/(T/2)^2)
m2 = -(\mu/(T/2)^2)

# == Formulate as an LQ problem == #
Q = 1.0
R = zeros(4, 4)
Rf = zeros(4, 4); Rf[1, 1] = q
A = [1 + r -c_bar m1 m2; 0 1 0 0; 0 1 1 0; 0 1 2 1]
B = [-1.0; 0.0; 0.0; 0.0]
C = [\sigma; 0.0; 0.0; 0.0]

# == Compute solutions and simulate == #
lq = LQ(Q, R, A, B, C; bet=\beta, capT=T, rf=Rf)
x0 = [0.0; 1.0; 0.0; 0.0]
xp, up, wp = compute_sequence(lq, x0)

# == Convert results back to assets, consumption and income == #
ap = vec(xp[1, 1:end])  # Assets
c = vec(up + c_bar)  # Consumption
time = 1:T
income = \sigma * vec(wp[1, 2:end]) + m1 * time + m2 * time.^2  # Income

# == Plot results == #
p1 = plot(Vector[income, ap, c, zeros(T+1)],
```

474 Chapter 3. Dynamic Programming
Exercise 2

This is a permanent income / life-cycle model with polynomial growth in income over working life followed by a fixed retirement income. The model is solved by combining two LQ programming problems as described in the lecture.

```julia
# == Model parameters == #
r = 0.05
β = 1/(1 + r)
```
\[
T = 60 \\
K = 40 \\
c\_bar = 4 \\
\sigma = 0.35 \\
\mu = 4 \\
q = 1e4 \\
s = 1 \\
m1 = 2 * \mu / K \\
m2 = - \mu / K^2
\]

# == Formulate LQ problem 1 (retirement) == #
\[
Q = 1.0 \\
R = zeros(4, 4) \\
Rf = zeros(4, 4); Rf[1, 1] = q \\
A = [1+r s-c\_bar 0 0; 0 1 0 0; 0 1 1 0; 0 1 2 1] \\
B = [-1.0; 0.0; 0.0; 0.0] \\
C = [0.0; 0.0; 0.0; 0.0]
\]

# == Initialize LQ instance for retired agent == #
lq\_retired = LQ(Q, R, A, B, C; bet=\beta, capT=T-K, rf=Rf) 
lq\_retired\_proxy = LQ(Q, R, A, B, C; bet=\beta, capT=T-K, rf=Rf) # Since update\_values!() changes its argument in place, we need another identical instance just to get the correct value function

# == Iterate back to start of retirement, record final value function == #
for i in 1:(T-K)
    update\_values!(lq\_retired\_proxy)
end
Rf2 = lq\_retired\_proxy.P

# == Formulate LQ problem 2 (working life) == #
\[
Q = 1.0 \\
R = zeros(4, 4) \\
A = [1 + r -c\_bar m1 m2; 0 1 0 0; 0 1 1 0; 0 1 2 1] \\
B = [-1.0; 0.0; 0.0; 0.0] \\
C = [\sigma; 0.0; 0.0; 0.0]
\]

# == Set up working life LQ instance with terminal Rf from lq\_retired == #
lq\_working = LQ(Q, R, A, B, C; bet=\beta, capT=K, rf=Rf2)

# == Simulate working state / control paths == #
x0 = [0.0; 1.0; 0.0; 0.0]
xp_w, up_w, wp_w = compute_sequence(lq_working, x0)
# == Simulate retirement paths (note the initial condition) ==#
xp_r, up_r, wp_r = compute_sequence(lq_retired, xp_w[:, end])

# == Convert results back to assets, consumption and income ==#
xp = [xp_w xp_r[:, 2:end]]
assets = vec(xp[1, :])  # Assets
up = [up_w up_r]
c = vec(up + c_bar)  # Consumption
time = 1:K
income_w = σ * vec(wp_w[1, 2:K+1]) + m1 .* time + m2 .* time.^2  # Income
income_r = ones(T-K) * s
income = [income_w; income_r]

# == Plot results ==#
p2 = plot(Vector[income, assets, c, zeros(T + 1)],
         lab="non-financial income" "assets" "consumption" "")
         color=[:orange :blue :green :black],
xaxis="Time", layout=(2, 1),
         bottom_margin=20mm, size=(600, 600))
Exercise 3

The first task is to find the matrices $A, B, C, Q, R$ that define the LQ problem

Recall that $x_t = (\tilde{q}_t, q_t, 1)'$, while $u_t = q_{t+1} - q_t$

Letting $m_0 := (a_0 - c)/2a_1$ and $m_1 := 1/2a_1$, we can write $\tilde{q}_t = m_0 + m_1 d_t$, and then, with some manipulation

$$\tilde{q}_{t+1} = m_0 (1 - \rho) + \rho \tilde{q}_t + m_1 \sigma w_{t+1}$$

By our definition of $u_t$, the dynamics of $q_t$ are $q_{t+1} = q_t + u_t$

Using these facts you should be able to build the correct $A, B, C$ matrices (and then check them against those found in the solution code below)
Suitable $R, Q$ matrices can be found by inspecting the objective function, which we repeat here for convenience:

$$\min \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t a_1 (q_t - \bar{q}_t)^2 + \gamma u_t^2 \right\}$$

Our solution code is

```julia
# == Model parameters == #
a0 = 5.0
a1 = 0.5
σ = 0.15
ρ = 0.9
γ = 1.0
β = 0.95
c = 2.0
T = 120

# == Useful constants == #
m0 = (a0-c)/(2 * a1)
m1 = 1/(2 * a1)

# == Formulate LQ problem == #
Q = γ
R = [a1 -a1 0;
    -a1 a1 0;
    0 0 0]
A = [ρ 0 m0*(1-ρ);
    0 1 0;
    0 0 1]
B = [0.0; 1.0; 0.0]
C = [m1 + σ; 0.0; 0.0]
lq = LQ(Q, R, A, B, C; bet=β)

# == Simulate state / control paths == #
x0 = [m0; 2.0; 1.0]
xp, up, wp = compute_sequence(lq, x0, 150)
q_bar = vec(xp[1, :])
q = vec(xp[2, :])

# == Plot simulation results == #
p3 = plot(1:length(q), [q_bar q],
    lab=["q_bar" "q"],
    color=[:black :blue],
xaxis="Time", title="Dynamics with $\gamma = $γ",
    bottom_margin=20mm, top_margin=10mm,
    size=(700, 500))
```

3.11. LQ Dynamic Programming Problems
3.12 Optimal Savings I: The Permanent Income Model

Contents

- Optimal Savings I: The Permanent Income Model
  - Overview
  - The Savings Problem
  - Alternative Representations
  - Two Classic Examples
  - Further Reading
  - Appendix: the Euler Equation
3.12.1 Overview

This lecture describes a rational expectations version of the famous permanent income model of Milton Friedman [Fri56].

Robert Hall cast Friedmans model within a linear-quadratic setting [Hal78].

Like Hall, we formulate an infinite-horizon linear-quadratic savings problem. We use the model as a vehicle for illustrating

• alternative formulations of the state of a dynamic system
• the idea of cointegration
• impulse response functions
• the idea that changes in consumption are useful as predictors of movements in income

Background readings on the linear-quadratic-Gaussian permanent income model are Halls [Hal78] and chapter 2 of [LS18].

3.12.2 The Savings Problem

In this section we state and solve the savings and consumption problem faced by the consumer.

Preliminaries

We use a class of stochastic processes called martingales. A discrete time martingale is a stochastic process (i.e., a sequence of random variables) \( \{ X_t \} \) with finite mean at each \( t \) and satisfying

\[
\mathbb{E}_t[X_{t+1}] = X_t, \quad t = 0, 1, 2, \ldots
\]

Here \( \mathbb{E}_t := \mathbb{E}[\cdot | \mathcal{F}_t] \) is a conditional mathematical expectation conditional on the time \( t \) information set \( \mathcal{F}_t \).

The latter is just a collection of random variables that the modeler declares to be visible at \( t \).

• When not explicitly defined, it is usually understood that \( \mathcal{F}_t = \{ X_t, X_{t-1}, \ldots, X_0 \} \)

Martingales have the feature that the history of past outcomes provides no predictive power for changes between current and future outcomes.

For example, the current wealth of a gambler engaged in a fair game has this property.

One common class of martingales is the family of random walks. A random walk is a stochastic process \( \{ X_t \} \) that satisfies

\[
X_{t+1} = X_t + w_{t+1}
\]

for some iid zero mean innovation sequence \( \{ w_t \} \).
Evidently $X_t$ can also be expressed as

$$X_t = \sum_{j=1}^{t} w_j + X_0$$

Not every martingale arises as a random walk (see, for example, Wald’s martingale)

### The Decision Problem

A consumer has preferences over consumption streams that are ordered by the utility functional

$$\mathbb{E}_0 \left[ \sum_{t=0}^{\infty} \beta^t u(c_t) \right]$$

(3.89)

where

- $\mathbb{E}_t$ is the mathematical expectation conditioned on the consumers 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 (3.89) by choosing a consumption, borrowing plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t \quad t \geq 0$$

(3.90)

Here

- $y_t$ is an exogenous endowment process
- $r > 0$ is a time-invariant risk-free net interest rate
- $b_t$ is one-period risk-free debt maturing at $t$

The consumer also faces initial conditions $b_0$ and $y_0$, which can be fixed or random

### Assumptions

For the remainder of this lecture, we follow Friedman and Hall in assuming that $(1 + r)^{-1} = \beta$

Regarding the endowment process, we assume it has the state-space representation

$$z_{t+1} = Az_t + Cw_{t+1}$$
$$y_t = Uz_t$$

(3.91)
• \( \{w_t\} \) is an iid vector process with \( \mathbb{E}w_t = 0 \) and \( \mathbb{E}w_t w_t' = I \)
• the \textit{spectral radius} of \( A \) satisfies \( \rho(A) < \sqrt{1/\beta} \)
• \( U \) is a selection vector that pins down \( y_t \) as a particular linear combination of components of \( z_t \).

The restriction on \( \rho(A) \) prevents income from growing so fast that discounted geometric sums of some quadratic forms to be described below become infinite.

Regarding preferences, we assume the quadratic utility function
\[
u(c_t) = -(c_t - \gamma)^2
\]
where \( \gamma \) is a bliss level of consumption.

\textbf{Note:} Along with this quadratic utility specification, we allow consumption to be negative. However, by choosing parameters appropriately, we can make the probability that the model generates negative consumption paths over finite time horizons as low as desired.

Finally, we impose the \textit{no Ponzi scheme} condition
\[
\mathbb{E}_0 \left[ \sum_{t=0}^{\infty} \beta^t b_t^2 \right] < \infty
\] (3.92)
This condition rules out an always-borrow scheme that would allow the consumer to enjoy bliss consumption forever.

\textbf{First-Order Conditions}

First-order conditions for maximizing (3.89) subject to (3.90) are
\[
\mathbb{E}_t[u'(c_{t+1})] = u'(c_t), \quad t = 0, 1, \ldots
\] (3.93)
These optimality conditions are also known as \textit{Euler equations}.

If you’re not sure where they come from, you can find a proof sketch in the \textit{appendix}.

With our quadratic preference specification, (3.93) has the striking implication that consumption follows a martingale:
\[
\mathbb{E}_t[c_{t+1}] = c_t
\] (3.94)
(In fact quadratic preferences are \textit{necessary} for this conclusion\(^1\))

\(^1\) A linear marginal utility is essential for deriving (3.94) from (3.93). Suppose instead that we had imposed the following more standard assumptions on the utility function: \( u'(c) > 0, u''(c) < 0, u'''(c) > 0 \) and required that \( c \geq 0 \). The Euler equation remains (3.93). But the fact that \( u''' < 0 \) implies via Jenseins inequality that \( \mathbb{E}_t[u'(c_{t+1})] > u' (\mathbb{E}_t[c_{t+1}]) \). This inequality together with (3.93) implies that \( \mathbb{E}_t[c_{t+1}] > c_t \) (consumption is said to be a submartingale), so that consumption stochastically diverges to \(+\infty\). The consumers savings also diverge to \(+\infty\).
One way to interpret (3.94) is that consumption will change only when new information about permanent income is revealed.

These ideas will be clarified below.

**The Optimal Decision Rule**

Now let's deduce the optimal decision rule.

**Note:** One way to solve the consumer's problem is to apply dynamic programming as in this lecture. We do this later. But first, we use an alternative approach that is revealing and shows the work that dynamic programming does for us behind the scenes.

In doing so, we need to combine:

1. the optimality condition (3.94)
2. the period-by-period budget constraint (3.90), and
3. the boundary condition (3.92)

To accomplish this, observe first that (3.92) implies $\lim_{t \to \infty} \beta^{1/2} b_{t+1} = 0$.

Using this restriction on the debt path and solving (3.90) forward yields

$$b_t = \sum_{j=0}^{\infty} \beta^j (y_{t+j} - c_{t+j})$$

(3.95)

Take conditional expectations on both sides of (3.95) and use the martingale property of consumption and the law of iterated expectations to deduce

$$b_t = \sum_{j=0}^{\infty} \beta^j E_t[y_{t+j}] - \frac{c_t}{1 - \beta}$$

(3.96)

Expressed in terms of $c_t$, we get

$$c_t = (1 - \beta) \left[ \sum_{j=0}^{\infty} \beta^j E_t[y_{t+j}] - b_t \right] = \frac{r}{1 + r} \left[ \sum_{j=0}^{\infty} \beta^j E_t[y_{t+j}] - b_t \right]$$

(3.97)

where the last equality uses $(1 + r)\beta = 1$.

These last two equations assert that consumption equals *economic income*

- **financial wealth** equals $-b_t$
• **non-financial wealth** equals $\sum_{j=0}^{\infty} \beta^j E_t[y_{t+j}]$

• **total wealth** equals the sum of financial and non-financial wealth

• A **marginal propensity to consume out of total wealth** equals the interest factor $\frac{r}{1+r}$

• **economic income** equals
  
  – a constant marginal propensity to consume times the sum of non-financial wealth and financial wealth
  
  – the amount the consumer can consume while leaving its wealth intact

### Responding to the State

The *state* vector confronting the consumer at $t$ is $[b_t \quad z_t]$

Here

• $z_t$ is an *exogenous* component, unaffected by consumer behavior

• $b_t$ is an *endogenous* component (since it depends on the decision rule)

Note that $z_t$ contains all variables useful for forecasting the consumers future endowment

It is plausible that current decisions $c_t$ and $b_{t+1}$ should be expressible as functions of $z_t$ and $b_t$

This is indeed the case

In fact, from *this discussion* we see that

$$\sum_{j=0}^{\infty} \beta^j E_t[y_{t+j}] = E_t \left[ \sum_{j=0}^{\infty} \beta^j y_{t+j} \right] = U(I - \beta A)^{-1} z_t$$

Combining this with (3.97) gives

$$c_t = \frac{r}{1+r} \left[ U(I - \beta A)^{-1} z_t - b_t \right] \quad (3.98)$$

Using this equality to eliminate $c_t$ in the budget constraint (3.90) gives

$$b_{t+1} = (1+r)(b_t + c_t - y_t)$$

$$= (1+r)b_t + r[U(I - \beta A)^{-1} z_t - b_t] - (1+r)U z_t$$

$$= b_t + U[r(I - \beta A)^{-1} - (1+r)I] z_t$$

$$= b_t + U(I - \beta A)^{-1}(A - I) z_t$$

To get from the second last to the last expression in this chain of equalities is not trivial

A key is to use the fact that $(1+r)\beta = 1$ and $(I - \beta A)^{-1} = \sum_{j=0}^{\infty} \beta^j A^j$

We've now successfully written $c_t$ and $b_{t+1}$ as functions of $b_t$ and $z_t$
A State-Space Representation

We can summarize our dynamics in the form of a linear state-space system governing consumption, debt and income:

\[
\begin{align*}
    z_{t+1} &= Az_t + Cw_{t+1} \\
    b_{t+1} &= b_t + U[(I - \beta A)^{-1}(A - I)]z_t \\
    y_t &= Uz_t \\
    c_t &= (1 - \beta)[U(I - \beta A)^{-1}z_t - b_t]
\end{align*}
\]  

(3.99)

To write this more succinctly, let

\[
    x_t = \begin{bmatrix} z_t \\ b_t \end{bmatrix}, \quad \tilde{A} = \begin{bmatrix} A & 0 \\ U(I - \beta A)^{-1}(A - I) & 1 \end{bmatrix}, \quad \tilde{C} = \begin{bmatrix} C' \\ 0 \end{bmatrix}
\]

and

\[
    \tilde{U} = \begin{bmatrix} U \\ (1 - \beta)U(I - \beta A)^{-1} - (1 - \beta) \end{bmatrix}, \quad \tilde{\eta}_t = \begin{bmatrix} y_t \\ c_t \end{bmatrix}
\]

Then we can express equation (3.99) as

\[
\begin{align*}
    x_{t+1} &= \tilde{A}x_t + \tilde{C}w_{t+1} \\
    \tilde{\eta}_t &= \tilde{U}x_t
\end{align*}
\]  

(3.100)

We can use the following formulas from linear state space models to compute population mean \( \mu_t = \mathbb{E}x_t \) and covariance \( \Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)'] \)

\[
\begin{align*}
    \mu_{t+1} &= \tilde{A}\mu_t \quad \text{with} \quad \mu_0 \text{ given} \quad (3.101) \\
    \Sigma_{t+1} &= \tilde{A}\Sigma_t\tilde{A}^T + \tilde{C}\tilde{C}' \quad \text{with} \quad \Sigma_0 \text{ given} \quad (3.102)
\end{align*}
\]

We can then compute the mean and covariance of \( \tilde{\eta}_t \) from

\[
\begin{align*}
    \mu_{y,t} &= \tilde{U}\mu_t \\
    \Sigma_{y,t} &= \tilde{U}\Sigma_t\tilde{U}'
\end{align*}
\]  

(3.103)

A Simple Example with iid Income

To gain some preliminary intuition on the implications of (3.99), lets look at a highly stylized example where income is just iid.
(Later examples will investigate more realistic income streams)

In particular, let \( \{w_t\}_{t=1}^{\infty} \) be iid and scalar standard normal, and let

\[
\begin{bmatrix}
z_t \\
1
\end{bmatrix}, \quad A = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & \mu \end{bmatrix}, \quad C = \begin{bmatrix} \sigma \\ 0 \end{bmatrix}
\]

Finally, let \( b_0 = z_0 = 0 \)

Under these assumptions we have \( y_t = \mu + \sigma w_t \sim N(\mu, \sigma^2) \)

Further, if you work through the state space representation, you will see that

\[
b_t = -\sigma \sum_{j=1}^{t-1} w_j \\
c_t = \mu + (1 - \beta)\sigma \sum_{j=1}^{t} w_j
\]

Thus income is iid and debt and consumption are both Gaussian random walks

Defining assets as \( -b_t \), we see that assets are just the cumulative sum of unanticipated incomes prior to the present date

The next figure shows a typical realization with \( r = 0.05, \mu = 1, \) and \( \sigma = 0.15 \)

```julia
#=
@author: Spencer Lyon Victoria Gregory
#

using Plots
pyplot()

const r = 0.05
const \beta = 1.0 / (1.0 + r)
const T = 60
const \sigma = 0.15
const \mu = 1.0

function time_path2()
    w = randn(T+1)
    w[1] = 0.0
    b = Array(Float64)(T+1)
    for t=2:T+1
        b[t] = sum(w[1:t])
    end
    b .*=-\sigma
    c = \mu + (1.0 - \beta) .* (\sigma .* w .- b)
    return w, b, c
end
```
Observe that consumption is considerably smoother than income.

The figure below shows the consumption paths of 250 consumers with independent income streams.

```julia
w, b, c = time_path2()
p = plot(0:T, μ + σ .* w, color=:green, label="non-financial income")
plot!(c, color=:black, label="consumption")
plot!(b, color=:blue, label="debt")
plot!(xlabel="Time", linewidth=2, alpha=0.7, xlims=(0, T))
```

```julia
time_paths = []
n = 250

for i=1:n
    push!(time_paths, time_path2()[3])
end

p = plot(time_paths, linewidth=0.8, alpha=0.7, legend=:none)
plot!(xlabel="Time", ylabel="Consumption", xlims=(0, T))
```
3.12.3 Alternative Representations

In this section we shed more light on the evolution of savings, debt and consumption by representing their dynamics in several different ways.

Halls Representation

Hall [Hal78] suggested an insightful way to summarize the implications of LQ permanent income theory. First, to represent the solution for $b_t$, shift (3.97) forward one period and eliminate $b_{t+1}$ by using (3.90) to obtain

$$c_{t+1} = (1 - \beta) \sum_{j=0}^{\infty} \beta^j \mathbb{E}_{t+1}[y_{t+j+1}] - (1 - \beta) \left[ \beta^{-1} (c_t + b_t - y_t) \right]$$

If we add and subtract $\beta^{-1} (1 - \beta) \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t y_{t+j}$ from the right side of the preceding equation and rearrange, we obtain

$$c_{t+1} - c_t = (1 - \beta) \sum_{j=0}^{\infty} \beta^j \{ \mathbb{E}_{t+1}[y_{t+j+1}] - \mathbb{E}_t[y_{t+j+1}] \}$$

(3.104)

The right side is the time $t+1$ innovation to the expected present value of the endowment process $\{y_t\}$. We can represent the optimal decision rule for $(c_t, b_{t+1})$ in the form of (3.104) and (3.96), which we repeat:
Equation (3.105) asserts that the consumers’ debt due at \( t \) equals the expected present value of its endowment minus the expected present value of its consumption stream.

A high debt thus indicates a large expected present value of surpluses \( y_t - c_t \).

Recalling again our discussion on forecasting geometric sums, we have

\[
\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} = U(I - \beta A)^{-1} z_t
\]

\[
\mathbb{E}_{t+1} \sum_{j=0}^{\infty} \beta^j y_{t+j+1} = U(I - \beta A)^{-1} z_{t+1}
\]

\[
\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j+1} = U(I - \beta A)^{-1} Az_t
\]

Using these formulas together with (3.91) and substituting into (3.104) and (3.105) gives the following representation for the consumers’ optimum decision rule:

\[
c_{t+1} = c_t + (1 - \beta) U(I - \beta A)^{-1} Cw_{t+1}
\]

\[
b_t = U(I - \beta A)^{-1} z_t - \frac{1}{1 - \beta} c_t
\]

\[
y_t = U z_t
\]

\[
z_{t+1} = Az_t + Cw_{t+1}
\]

Representation (3.106) makes clear that

- The state can be taken as \((c_t, z_t)\)
  - The endogenous part is \(c_t\) and the exogenous part is \(z_t\)
  - Debt \(b_t\) has disappeared as a component of the state because it is encoded in \(c_t\)
- Consumption is a random walk with innovation \((1 - \beta) U(I - \beta A)^{-1} Cw_{t+1}\)
  - This is a more explicit representation of the martingale result in (3.94)

**Cointegration**

Representation (3.106) reveals that the joint process \(\{c_t, b_t\}\) possesses the property that Engle and Granger [EG87] called cointegration.

Cointegration is a tool that allows us to apply powerful results from the theory of stationary stochastic processes to (certain transformations of) nonstationary models.
To apply cointegration in the present context, suppose that $z_t$ is asymptotically stationary\(^4\).

Despite this, both $c_t$ and $b_t$ will be non-stationary because they have unit roots (see (3.99) for $b_t$).

Nevertheless, there is a linear combination of $c_t, b_t$ that is asymptotically stationary.

In particular, from the second equality in (3.106) we have

$$
(1 - \beta)b_t + c_t = (1 - \beta)U(I - \beta A)^{-1}z_t \tag{3.107}
$$

Hence the linear combination $(1 - \beta)b_t + c_t$ is asymptotically stationary.

Accordingly, Granger and Engle would call $[(1 - \beta) \quad 1]$ a cointegrating vector for the state.

When applied to the nonstationary vector process $[b_t \quad c_t]'$, it yields a process that is asymptotically stationary.

Equation (3.107) can be rearranged to take the form

$$
(1 - \beta)b_t + c_t = (1 - \beta)E_t \sum_{j=0}^{\infty} \beta^j y_{t+j}. \tag{3.108}
$$

Equation (3.108) asserts that the cointegrating residual on the left side equals the conditional expectation of the geometric sum of future incomes on the right\(^6\).

**Cross-Sectional Implications**

Consider again (3.106), this time in light of our discussion of distribution dynamics in the lecture on linear systems.

The dynamics of $c_t$ are given by

$$
c_{t+1} = c_t + (1 - \beta)U(I - \beta A)^{-1}C w_{t+1} \tag{3.109}
$$

or

$$
c_t = c_0 + \sum_{j=1}^{t} \tilde{w}_j \quad \text{for} \quad \tilde{w}_{t+1} := (1 - \beta)U(I - \beta A)^{-1}C w_{t+1}
$$

The unit root affecting $c_t$ causes the time $t$ variance of $c_t$ to grow linearly with $t$.

In particular, since $\{\tilde{w}_t\}$ is iid, we have

$$
\text{Var}[c_t] = \text{Var}[c_0] + t \sigma^2 \tag{3.110}
$$

---

\(^4\) This would be the case if, for example, the spectral radius of $A$ is strictly less than one.

\(^6\) See [JYC88], [LL01], [LL04] for interesting applications of related ideas.
where
\[ \hat{\sigma}^2 := (1 - \beta)^2 U(I - \beta A)^{-1} C C'(I - \beta A')^{-1} U' \]

When \( \hat{\sigma} > 0 \), \( \{c_t\} \) has no asymptotic distribution

Let's consider what this means for a cross-section of ex ante identical consumers born at time 0

Let the distribution of \( c_0 \) represent the cross-section of initial consumption values

Equation (3.110) tells us that the variance of \( c_t \) increases over time at a rate proportional to \( t \)

A number of different studies have investigated this prediction and found some support for it (see, e.g., [DP94], [STY04])

**Impulse Response Functions**

Impulse response functions measure responses to various impulses (i.e., temporary shocks)

The impulse response function of \( \{c_t\} \) to the innovation \( \{w_t\} \) is a box

In particular, the response of \( c_{t+j} \) to a unit increase in the innovation \( w_{t+1} \) is \( (1 - \beta)U(I - \beta A)^{-1}C \) for all \( j \geq 1 \)

**Moving Average Representation**

It's useful to express the innovation to the expected present value of the endowment process in terms of a moving average representation for income \( y_t \)

The endowment process defined by (3.91) has the moving average representation

\[ y_{t+1} = d(L) w_{t+1} \]  \hspace{1cm} (3.111)

where
- \( d(L) = \sum_{j=0}^{\infty} d_j L^j \) for some sequence \( d_j \), where \( L \) is the lag operator
- at time \( t \), the consumer has an information set \( w^t = [w_t, w_{t-1}, \ldots] \)

Notice that
\[ y_{t+j} - \mathbb{E}_t[y_{t+j}] = d_0 w_{t+j} + d_1 w_{t+j-1} + \cdots + d_{j-1} w_{t+1} \]

It follows that
\[ \mathbb{E}_{t+1}[y_{t+j}] - \mathbb{E}_t[y_{t+j}] = d_{j-1} w_{t+1} \]  \hspace{1cm} (3.112)

\(^3\) Representation (3.91) implies that \( d(L) = U(I - AL)^{-1}C \).

\(^5\) A moving average representation for a process \( y_t \) is said to be **fundamental** if the linear space spanned by \( y^t \) is equal to the linear space spanned by \( w^t \). A time-invariant innovations representation, attained via the Kalman filter, is by construction fundamental.
Using (3.112) in (3.104) gives

\[ c_{t+1} - c_t = (1 - \beta)d(\beta)w_{t+1} \]  

(3.113)

The object \( d(\beta) \) is the **present value of the moving average coefficients** in the representation for the endowment process \( y_t \).

### 3.12.4 Two Classic Examples

We illustrate some of the preceding ideas with two examples.

In both examples, the endowment follows the process \( y_t = z_{1t} + z_{2t} \) where

\[
\begin{bmatrix}
    z_{1t+1} \\
    z_{2t+1}
\end{bmatrix} =
\begin{bmatrix}
    1 & 0 \\
    0 & 0
\end{bmatrix}
\begin{bmatrix}
    z_{1t} \\
    z_{2t}
\end{bmatrix} +
\begin{bmatrix}
    \sigma_1 & 0 \\
    0 & \sigma_2
\end{bmatrix}
\begin{bmatrix}
    w_{1t+1} \\
    w_{2t+1}
\end{bmatrix}
\]

Here

- \( w_{t+1} \) is an iid \( 2 \times 1 \) process distributed as \( N(0, I) \)
- \( z_{1t} \) is a permanent component of \( y_t \)
- \( z_{2t} \) is a purely transitory component of \( y_t \)

**Example 1**

Assume as before that the consumer observes the state \( z_t \) at time \( t \).

In view of (3.106) we have

\[ c_{t+1} - c_t = \sigma_1 w_{1t+1} + (1 - \beta)\sigma_2 w_{2t+1} \]  

(3.114)

Formula (3.114) shows how an increment \( \sigma_1 w_{1t+1} \) to the permanent component of income \( z_{1t+1} \) leads to

- a permanent one-for-one increase in consumption and
- no increase in savings \( -b_{t+1} \)

But the purely transitory component of income \( \sigma_2 w_{2t+1} \) leads to a permanent increment in consumption by a fraction \( 1 - \beta \) of transitory income.

The remaining fraction \( \beta \) is saved, leading to a permanent increment in \( -b_{t+1} \).

Application of the formula for debt in (3.99) to this example shows that

\[ b_{t+1} - b_t = -z_{2t} = -\sigma_2 w_{2t} \]  

(3.115)

This confirms that none of \( \sigma_1 w_{1t} \) is saved, while all of \( \sigma_2 w_{2t} \) is saved.

The next figure illustrates these very different reactions to transitory and permanent income shocks using impulse-response functions.

const \( r = 0.05 \)
const \( \beta = 1.0 / (1.0 + r) \)
const \( T = 20 \)  # Time horizon
const \( S = 5 \)  # Impulse date
const \( \sigma_1 = 0.15 \)
const \( \sigma_2 = 0.15 \)

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] - \sigma_2 \times w2[t]
        c[t+1] = c[t] + \sigma_1 \times w1[t+1] + (1 - \beta) \times \sigma_2 \times w2[t+1]
    end

    return b, c
end

L = 0.175

b1, c1 = time_path(false)
b2, c2 = time_path(true)
p = plot(0:T, [c1 c2 b1 b2], layout=(2, 1),
    color=[:green :green :blue :blue],
    label=["consumption" "consumption" "debt" "debt"])
t = ["impulse-response, transitory income shock"
    "impulse-response, permanent income shock"]
plot!(title=reshape(t,1,length(t)), xlabel="Time", ylims=(-L, L),
    legend=["topright :bottomright"]
vline!([S S], color=:black, layout=(2, 1), label=""))
Example 2

Assume now that at time $t$ the consumer observes $y_t$, and its history up to $t$, but not $z_t$.

Under this assumption, it is appropriate to use an *innovation representation* to form $A, C, U$ in (3.106).

The discussion in sections 2.9.1 and 2.11.3 of [LS18] shows that the pertinent state space representation for $y_t$ is

$$
\begin{bmatrix}
  y_{t+1} \\
  a_{t+1}
\end{bmatrix} = \begin{bmatrix}
  1 & -(1 - K) \\
  0 & 0
\end{bmatrix} \begin{bmatrix}
  y_t \\
  a_t
\end{bmatrix} + \begin{bmatrix}
  1 \\
  1
\end{bmatrix} a_{t+1}
$$

where

- $K :=$ the stationary Kalman gain
- $a_t := y_t - E[y_t | y_{t-1}, \ldots, y_0]$

In the same discussion in [LS18] it is shown that $K \in [0, 1]$ and that $K$ increases as $\sigma_1/\sigma_2$ does.
In other words, $K$ increases as the ratio of the standard deviation of the permanent shock to that of the transitory shock increases

Please see *first look at the Kalman filter*

Applying formulas (3.106) implies

\[
ct_{t+1} - ct = [1 - \beta(1 - K)]at_{t+1}
\]

(3.116)

where the endowment process can now be represented in terms of the univariate innovation to $yt$ as

\[
yt_{t+1} - yt = at_{t+1} - (1 - K)at
\]

(3.117)

Equation (3.117) indicates that the consumer regards

- fraction $K$ of an innovation $at_{t+1}$ to $yt_{t+1}$ 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 $at_{t+1}$, but by only $(1 - \beta)$ times his estimate of the purely transitory part of $at_{t+1}$.

Therefore, in total he permanently increments his consumption by a fraction $K + (1 - \beta)(1 - K) = 1 - \beta(1 - K)$ of $at_{t+1}$.

He saves the remaining fraction $\beta(1 - K)$.

According to equation (3.117), the first difference of income is a first-order moving average.

Equation (3.116) asserts that the first difference of consumption is iid.

Application of formula to this example shows that

\[
b_{t+1} - bt = (K - 1)at
\]

(3.118)

This indicates how the fraction $K$ of the innovation to $yt$ that is regarded as permanent influences the fraction of the innovation that is saved.

### 3.12.5 Further Reading

The model described above significantly changed how economists think about consumption.

While Hall’s model does a remarkably good job as a first approximation to consumption data, it is widely believed that it doesn’t capture important aspects of some consumption/savings data.

For example, liquidity constraints and precautionary savings appear to be present sometimes.

Further discussion can be found in, e.g., [HM82], [Par99], [Dea91], [Car01]
3.12.6 Appendix: the Euler Equation

Where does the first order condition (3.93) come from?

Here we will 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 (3.90) we then have

$$c_0 = \frac{b_1}{1 + r} - b_0 + y_0 \quad \text{and} \quad c_1 = y_1 - b_1$$

Here $b_0$ and $y_0$ are given constants.

Substituting these constraints into our two period objective $u(c_0) + \beta \mathbb{E}_0[u(c_1)]$ gives

$$\max_{b_1} \left\{ u \left( \frac{b_1}{R} - b_0 + y_0 \right) + \beta \mathbb{E}_0[u(y_1 - b_1)] \right\}$$

You will be able to verify that the first order condition is

$$u'(c_0) = \beta R \mathbb{E}_0[u'(c_1)]$$

Using $\beta R = 1$ gives (3.93) in the two period case.

The proof for the general case is similar.

3.13 Optimal Savings II: LQ Techniques

Contents

- Optimal Savings II: LQ Techniques
  - Overview
  - Setup
  - The LQ Approach
  - Implementation
  - Two Example Economies

Co-author: Chase Coleman

3.13.1 Overview

This lecture continues our analysis of the linear-quadratic (LQ) permanent income model of savings and consumption.
As we saw in our previous lecture on this topic, Robert Hall [Hal78] used the LQ permanent income model to restrict and interpret intertemporal comovements of nondurable consumption, nonfinancial income, and financial wealth.

For example, we saw how the model asserts that for any covariance stationary process for nonfinancial income

- consumption is a random walk
- financial wealth has a unit root and is cointegrated with consumption

Other applications use the same LQ framework

For example, a model isomorphic to the LQ permanent income model has been used by Robert Barro [Bar79] to interpret intertemporal comovements of a governments tax collections, its expenditures net of debt service, and its public debt.

This isomorphism means that in analyzing the LQ permanent income model, we are in effect also analyzing the Barro tax smoothing model.

It is just a matter of appropriately relabeling the variables in Halls model.

In this lecture, well

- show how the solution to the LQ permanent income model can be obtained using LQ control methods
- represent the model as a linear state space system as in this lecture
- apply QuantEcon's LSS type to characterize statistical features of the consumers optimal consumption and borrowing plans

Well then use these characterizations to construct a simple model of cross-section wealth and consumption dynamics in the spirit of Truman Bewley [Bew86].

(Later well study other Bewley models see this lecture)

The model will prove useful for illustrating concepts such as

- stationarity
- ergodicity
- ensemble moments and cross section observations

### 3.13.2 Setup

Lets recall the basic features of the model discussed in permanent income model.

Consumer preferences are ordered by

$$E_0 \sum_{t=0}^{\infty} \beta^t u(c_t)$$

(3.119)

where $u(c) = -(c - \gamma)^2$
The consumer maximizes (3.119) by choosing a consumption, borrowing plan \( \{c_t, b_{t+1}\}_{t=0}^{\infty} \) subject to the sequence of budget constraints

\[
c_t + b_t = \frac{1}{1 + r} b_{t+1} + y_t, \quad t \geq 0
\]  

(3.120)

and the no-Ponzi condition

\[
E_0 \sum_{t=0}^{\infty} \beta^t b_t^2 < \infty
\]  

(3.121)

The interpretation of all variables and parameters are the same as in the previous lecture.

We continue to assume that \((1 + r)\beta = 1\).

The dynamics of \( \{y_t\} \) again follow the linear state space model

\[
z_{t+1} = Az_t + Cw_{t+1} \\
y_t = Uz_t
\]  

(3.122)

The restrictions on the shock process and parameters are the same as in our previous lecture.

**Digression on a useful isomorphism**

The LQ permanent income model of consumption is mathematically isomorphic with a version of Barros [Bar79] model of tax smoothing.

In the LQ permanent income model

- the household faces an exogenous process of nonfinancial income
- the household wants to smooth consumption across states and time

In the Barro tax smoothing model

- a government faces an exogenous sequence of government purchases (net of interest payments on its debt)
- a government wants to smooth tax collections across states and time

If we set

- \( T_t \), total tax collections in Barros model to consumption \( c_t \) in the LQ permanent income model
- \( G_t \), exogenous government expenditures in Barros model to nonfinancial income \( y_t \) in the permanent income model
- \( B_t \), government risk-free one-period assets falling due in Barros model to risk-free one period consumer debt \( b_t \) falling due in the LQ permanent income model
- \( R \), the gross rate of return on risk-free one-period government debt in Barros model to the gross rate of return \( 1 + r \) on financial assets in the permanent income model of consumption

3.13. Optimal Savings II: LQ Techniques 499
then the two models are mathematically equivalent

All characterizations of a \( \{c_t, y_t, b_t\} \) in the LQ permanent income model automatically apply to a \( \{T_t, G_t, B_t\} \) process in the Barro model of tax smoothing.

See consumption and tax smoothing models for further exploitation of an isomorphism between consumption and tax smoothing models.

### A specification of the nonfinancial income process

For the purposes of this lecture, let’s assume \( \{y_t\} \) is a second-order univariate autoregressive process:

\[
y_{t+1} = \alpha + \rho_1 y_t + \rho_2 y_{t-1} + \sigma w_{t+1}
\]

We can map this into the linear state space framework in (3.122), as discussed in our lecture on linear models.

To do so we take

\[
z_t = \begin{bmatrix} 1 & y_t & y_{t-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ \sigma \\ 0 \end{bmatrix}, \text{ and } U = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}
\]

### 3.13.3 The LQ Approach

Previously we solved the permanent income model by solving a system of linear expectational difference equations subject to two boundary conditions.

Here we solve the same model using LQ methods based on dynamic programming.

After confirming that answers produced by the two methods agree, we apply QuantEcon’s LSS type to illustrate features of the model.

Why solve a model in two distinct ways?

Because by doing so we gather insights about the structure of the model.

Our earlier approach based on solving a system of expectational difference equations brought to the fore the role of the consumers expectations about future nonfinancial income.

On the other hand, formulating the model in terms of an LQ dynamic programming problem reminds us that

- finding the state (of a dynamic programming problem) is an art, and
- iterations on a Bellman equation implicitly jointly solve both a forecasting problem and a control problem.

### The LQ Problem

Recall from our lecture on LQ theory that the optimal linear regulator problem is to choose a decision rule for \( u_t \) to minimize

\[
\mathbb{E} \sum_{t=0}^{\infty} \beta^t \{ x_t' R x_t + u_t' Q u_t \},
\]
subject to \( x_0 \) given and the law of motion

\[
x_{t+1} = \tilde{A}x_t + \tilde{B}u_t + \tilde{C}w_{t+1}, \quad t \geq 0,
\]

where \( w_{t+1} \) is iid with mean vector zero and \( \mathbb{E}w_tw'_t = I \)

The tildes in \( \tilde{A}, \tilde{B}, \tilde{C} \) are to avoid clashing with notation in \((3.122)\)

The value function for this problem is \( v(x) = -x'Px - d \), where

- \( P \) is the unique positive semidefinite solution of the corresponding matrix Riccati equation
- The scalar \( d \) is given by \( d = \beta(1 - \beta)^{-1}\text{trace}(P\tilde{C}\tilde{C}') \)

The optimal policy is \( u_t = -Fx_t \), where \( F := \beta(Q + \beta\tilde{B}'P\tilde{B})^{-1}\tilde{B}'P\tilde{A} \)

Under an optimal decision rule \( F \), the state vector \( x_t \) evolves according to \( x_{t+1} = (\tilde{A} - \tilde{B}F)x_t + \tilde{C}w_{t+1} \)

**Mapping into the LQ framework**

To map into the LQ framework, well use

\[
x_t := \begin{bmatrix} z_t \\ b_t \end{bmatrix} = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \\ b_t \end{bmatrix}
\]

as the state vector and \( u_t := c_t - \gamma \) as the control

With this notation and \( U_\gamma := \begin{bmatrix} \gamma & 0 \\ 0 & 0 \end{bmatrix} \), we can write the state dynamics as in \((3.123)\) when

\[
\tilde{A} := \begin{bmatrix} A \\ (1 + r)(U_\gamma - U) \end{bmatrix} \quad \tilde{B} := \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \text{and} \quad \tilde{C} := \begin{bmatrix} C' \\ 0 \end{bmatrix} w_{t+1}
\]

Please confirm for yourself that, with these definitions, the LQ dynamics \((3.123)\) match the dynamics of \( z_t \) and \( b_t \) described above

To map utility into the quadratic form \( x'_tRx_t + u'_tQu_t \) we can set

- \( Q := 1 \) (remember that we are minimizing) and
- \( R := a 4 \times 4 \) matrix of zeros

However, there is one problem remaining

We have no direct way to capture the non-recursive restriction \((3.121)\) on the debt sequence \( \{b_t\} \) from within the LQ framework

To try to enforce it, were going to use a trick: put a small penalty on \( b_t^2 \) in the criterion function

In the present setting, this means adding a small entry \( \epsilon > 0 \) in the \((4, 4)\) position of \( R \)

That will induce a (hopefully) small approximation error in the decision rule

Well check whether it really is small numerically soon

3.13. Optimal Savings II: LQ Techniques
3.13.4 Implementation

Let’s write some code to solve the model.

One comment before we start is that the bliss level of consumption $\gamma$ in the utility function has no effect on the optimal decision rule.

We saw this in the previous lecture *permanent income*.

The reason is that it drops out of the Euler equation for consumption.

In what follows we set it equal to unity.

**The exogenous nonfinancial income process**

First we create the objects for the optimal linear regulator.

```julia
using QuantEcon
using PyPlot

# Set parameters
$\alpha$, $\beta$, $\rho_1$, $\rho_2$, $\sigma$ = 10.0, 0.95, 0.9, 0.0, 1.0

$R = 1 / \beta$

$A = \begin{bmatrix}
1.0 & 0.0 & 0.0 \\
0.0 & \alpha & \rho_1 \rho_2 \\
0.0 & 1.0 & 0.0 
\end{bmatrix}$

$C = \begin{bmatrix} 0.0; \sigma; 0.0 \end{bmatrix}'$

$G = \begin{bmatrix} 0.0 & 1.0 & 0.0 \end{bmatrix}$

# Form LinearStateSpace system and pull off steady state moments

$\mu_z0 = \begin{bmatrix} 1.0, 0.0, 0.0 \end{bmatrix}$

$\Sigma_z0 = \text{zeros}(3, 3)$

$Lz = \text{LSS}(A, C, G, \mu_0=\mu_z0, \Sigma_0=\Sigma_z0)$

$\mu_z$, $\mu_y$, $\Sigma_z$, $\Sigma_y = \text{stationary_distributions}(Lz)$

# Mean vector of state for the savings problem

$mxo = [\mu_z; 0.0]$

# Create stationary covariance matrix of $x$ -- start everyone off at $b=0$

$a1 = \text{zeros}(3, 1)$

$aa = \text{hcat}(\Sigma_z, a1)$

$bb = \text{zeros}(1, 4)$

$sxo = \text{vcat}(aa, bb)$

# These choices will initialize the state vector of an individual at zero debt
# and the ergodic distribution of the endowment process. Use these to create
# the Bewley economy.

$mxbewley = mxo$

$sxbewley = sxo$
```

The next step is to create the matrices for the LQ system.
A12 = zeros(3,1)
ALQ_l = hcat(A, A12)
ALQ_r = [0 -R 0 R]
ALQ = vcat(ALQ_l, ALQ_r)

RLQ = [0.0 0.0 0.0 0.0;
      0.0 0.0 0.0 0.0;
      0.0 0.0 0.0 0.0;
      0.0 0.0 0.0 1e-9]

QLQ = 1.0
BLQ = [0.0; 0.0; 0.0; R]
CLQ = [0.0; σ; 0.0; 0.0]
β_LQ = β

Lets print these out and have a look at them

println("A = \$ALQ")
println("B = \$BLQ")
println("R = \$RLQ")
println("Q = \$QLQ")

A = [1.0 0.0 0.0 0.0; 10.0 0.9 0.0 0.0; 0.0 1.0 0.0 0.0; 0.0 -1.05263 0.0 1.0e-9]
B = [0.0, 0.0, 0.0, 1.05263]
R = [0.0 0.0 0.0 0.0; 0.0 0.0 0.0 0.0; 0.0 0.0 0.0 0.0; 0.0 0.0 0.0 1.0e-9]
Q = 1.0

Now create the appropriate instance of an LQ model

LQPI = LQ(QLQ, RLQ, ALQ, BLQ, CLQ, bet=β_LQ)

Well save the implied optimal policy function soon compare them with what we get by employing an alternative solution method

P, F, d = stationary_values(LQPI)  # Compute value function and decision rule
ABF = ALQ - BLQ * F  # Form closed loop system

**Comparison with the difference equation approach**

In our *first lecture* on the infinite horizon permanent income problem we used a different solution method

The method was based around

- deducing the Euler equations that are the first-order conditions with respect to consumption and savings
- using the budget constraints and boundary condition to complete a system of expectational linear difference equations
- solving those equations to obtain the solution
Expressed in state space notation, the solution took the form

\[
\begin{align*}
    z_{t+1} &= Az_t + Cw_{t+1} \\
    b_{t+1} &= b_t + U[(I - \beta A)^{-1}(A - I)]z_t \\
    y_t &= Uz_t \\
    c_t &= (1 - \beta)[U(I - \beta A)^{-1}z_t - b_t]
\end{align*}
\]

Now well apply the formulas in this system

```julia
# Use the above formulas to create the optimal policies for b_{t+1} and c_t
b_pol = G * (inv(eye(3, 3) - \beta * A)) * (A - eye(3, 3))
c_pol = (1 - \beta) * (G * inv(eye(3, 3) - \beta * A))

# Create the A matrix for a LinearStateSpace instance
A_LSS1 = vcat(A, b_pol)
A_LSS2 = [0, 0, 0, 1]
A_LSS = hcat(A_LSS1, A_LSS2)

# Create the C matrix for LSS methods
C_LSS = vcat(C, 0)

# Create the G matrix for LSS methods
G_LSS1 = vcat(G, c_pol)
G_LSS2 = vcat(0, -(1 - \beta))
G_LSS = hcat(G_LSS1, G_LSS2)

# Use the following values to start everyone off at b=0, initial incomes zero
μ₀ = [1.0, 0.0, 0.0, 0.0]
Σ₀ = zeros(4, 4)

A_LSS calculated as we have here should equal ABF calculated above using the LQ model

```

\[
\begin{array}{ccc}
0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
-9.51248e-6 & 9.51248e-8 & 0.0 & -2.0e-8
\end{array}
\]

Now compare pertinent elements of c_pol and F

```julia
println(c_pol, -F)
```

\[
\begin{bmatrix}
65.5172 & 0.344828 & 0.0 \\
65.5172 & 0.344828 & -0.0 & -0.05
\end{bmatrix}
\]

We have verified that the two methods give the same solution

Now lets create instances of the LSS type and use it to do some interesting experiments

To do this, well use the outcomes from our second method
3.13.5 Two Example Economies

In the spirit of Bewley models [Bew86], well generate panels of consumers. The examples differ only in the initial states with which we endow the consumers. All other parameter values are kept the same in the two examples:

- In the first example, all consumers begin with zero nonfinancial income and zero debt. The consumers are thus *ex ante* identical.
- In the second example, while all begin with zero debt, we draw their initial income levels from the invariant distribution of financial income. Consumers are *ex ante* heterogeneous.

In the first example, consumers nonfinancial income paths display pronounced transients early in the sample:

- these will affect outcomes in striking ways

Those transient effects will not be present in the second example.

We use methods affiliated with the LSS type to simulate the model.

**First set of initial conditions**

We generate 25 paths of the exogenous non-financial income process and the associated optimal consumption and debt paths.

In a first set of graphs, darker lines depict a particular sample path, while the lighter lines describe 24 other paths. A second graph plots a collection of simulations against the population distribution that we extract from the LSS instance LSS.

Comparing sample paths with population distributions at each date $t$ is a useful exercise; see our discussion of the laws of large numbers.

```plaintext
lss = LSS(A_LSS, C_LSS, G_LSS, mu_0=μ₀, Sigma_0=Σ₀)
```

**Population and sample panels**

In the code below, we use the LSS type to

- compute and plot population quantiles of the distributions of consumption and debt for a population of consumers
- simulate a group of 25 consumers and plot sample paths on the same graph as the population distribution

```plaintext
" * "
This function takes initial conditions $(μ₀, Σ₀)$ and uses the LSS type from QuantEcon to simulate an economy `npaths` times for `T` periods.
```

3.13. Optimal Savings II: LQ Techniques
It then uses that information to generate some graphs related to the discussion below. ""

```julia
function income_consumption_debt_series(A, C, G, μ₀, Σ₀, T=150, npaths=25)

    lss = LSS(A, C, G, μ₀=μ₀, Σ₀=Σ₀)

    # Simulation/Moment Parameters
    moment_generator = moment_sequence(lss)

    # Simulate various paths
    bsim = Array{Float64}(npaths, T)
    csim = Array{Float64}(npaths, T)
    ysim = Array{Float64}(npaths, T)
    for i = 1:npaths
        sims = simulate(lss, T)
        bsim[i, :) = sims[1][end, :]
        csim[i, :) = sims[2][2, :]
        ysim[i, :) = sims[2][1, :]
    end

    # Get the moments
    cons_mean = Array{Float64}(T)
    cons_var = Array{Float64}(T)
    debt_mean = Array{Float64}(T)
    debt_var = Array{Float64}(T)
    state = start(moment_generator)
    for t = 1:T
        (μₓ, μᵧ, Σₓ, Σᵧ), state = next(moment_generator, state)
        cons_mean[t], cons_var[t] = μᵧ[2], Σᵧ[2, 2]
        debt_mean[t], debt_var[t] = μₓ[4], Σₓ[4, 4]
    end
    return bsim, csim, ysim, cons_mean, cons_var, debt_mean, debt_var
end

function consumption_income_debt_figure(bsim, csim, ysim)

    # Get T
    T = size(bsim, 2)

    # Create first figure
    fig, ax = subplots(2, 1, figsize=(10, 8))
    xvals = 1:T

    # Plot consumption and income
    ax[1][][plot](csim[1, :], label="c", color="b")
    ax[1][][plot](ysim[1, :], label="y", color="g")
    ax[1][][plot](csim', alpha=.1, color="b")
    ax[1][][plot](ysim', alpha=.1, color="g")
    ax[1][][legend](loc=4)
    ax[1][][set](title="Nonfinancial Income, Consumption, and Debt",
```
```julia
xlabel="t", ylabel="y and c"

# Plot debt
ax[2][:plot](bsim[1, :], label="b", color="r")
ax[2][:plot](bsim', alpha=.1, color="r")
ax[2][:legend](loc=4)
ax[2][:set](xlabel="t", ylabel="debt")

fig[:tight_layout]

function consumption_debt_fanchart(csim, cons_mean, cons_var,
            bsim, debt_mean, debt_var)
    # Get T
    T = size(bsim, 2)

    # Create Percentiles of cross-section distributions
    cmean = mean(cons_mean)
    c90 = 1.65 * sqrt.(cons_var)
    c95 = 1.96 * sqrt.(cons_var)
    c_perc_95p, c_perc_95m = cons_mean + c95, cons_mean - c95
    c_perc_90p, c_perc_90m = cons_mean + c90, cons_mean - c90

    # Create Percentiles of cross-section distributions
    dmean = mean(debt_mean)
    d90 = 1.65 * sqrt.(debt_var)
    d95 = 1.96 * sqrt.(debt_var)
    d_perc_95p, d Perc_95m = debt_mean + d95, debt_mean - d95
    d_perc_90p, d_perc_90m = debt_mean + d90, debt_mean - d90

    # Create second figure
    fig, ax = subplots(2, 1, figsize=(10, 8))
    xvals = 1:T

    fig[:suplute]("Consumption/Debt over time")

    # Consumption fan
    ax[1][:plot](xvals, cons_mean, color="k")
    ax[1][:plot](xvals, csim', color="k", alpha=.25)
    ax[1][:fill_between](xvals, c_perc_95m, c_perc_95p, alpha=.25, color="b")
    ax[1][:fill_between](xvals, c_perc_90m, c_perc_90p, alpha=.25, color="r")
    ax[1][:set](title="Consumption/Debt over time",
               ylim=(cmean-15, cmean+15), ylabel="consumption")

    # Debt fan
    ax[2][:plot](xvals, debt_mean, color="k")
    ax[2][:plot](xvals, bsim', color="k", alpha=.25)
    ax[2][:fill_between](xvals, d_perc_95m, d_perc_95p, alpha=.25, color="b")
    ax[2][:fill_between](xvals, d_perc_90m, d_perc_90p, alpha=.25, color="r")
    ax[2][:set](ylablae="debt", xlabel="t")

    fig[:tight_layout]
end
```

3.13. Optimal Savings II: LQ Techniques 507
Now let's create figures with initial conditions of zero for $y_0$ and $b_0$

```julia
out = income_consumption_debt_series(A_LSS, C_LSS, G_LSS, μ₀, Σ₀)
bsim0, csim0, ysim0 = out[1:3]
cons_mean0, cons_var0, debt_mean0, debt_var0 = out[4:end]

consumption_income_debt_figure(bsim0, csim0, ysim0)

consumption_debt_fanchart(csim0, cons_mean0, cons_var0, bsim0, debt_mean0, debt_var0)
```
Here is what is going on in the above graphs.

For our simulation, we have set initial conditions $b_0 = y_{-1} = y_{-2} = 0$.

Because $y_{-1} = y_{-2} = 0$, nonfinancial income $y_t$ starts far below its stationary mean $\mu_{y,\infty}$ and rises early in each simulation.

Recall from the previous lecture that we can represent the optimal decision rule for consumption in terms of the co-integrating relationship

$$(1 - \beta)b_t + c_t = (1 - \beta)E_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$$ \hspace{1cm} (3.124)

So at time 0 we have

$$c_0 = (1 - \beta)E_0 \sum_{t=0}^{\infty} \beta^j y_t$$

This tells us that consumption starts at the income that would be paid by an annuity whose value equals the expected discounted value of nonfinancial income at time $t = 0$.

3.13. Optimal Savings II: LQ Techniques

509
To support that level of consumption, the consumer borrows a lot early and consequently builds up substantial debt.

In fact, he or she incurs so much debt that eventually, in the stochastic steady state, he consumes less each period than his nonfinancial income.

He uses the gap between consumption and nonfinancial income mostly to service the interest payments due on his debt.

Thus, when we look at the panel of debt in the accompanying graph, we see that this is a group of ex ante identical people each of whom starts with zero debt.

All of them accumulate debt in anticipation of rising nonfinancial income.

They expect their nonfinancial income to rise toward the invariant distribution of income, a consequence of our having started them at $y_{-1} = y_{-2} = 0$.

**Cointegration residual**

The following figure plots realizations of the left side of (3.124), which, as discussed in our last lecture, is called the cointegrating residual.

As mentioned above, the right side can be thought of as an annuity payment on the expected present value of future income $E_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$.

Early along a realization, $c_t$ is approximately constant while $(1 - \beta) b_t$ and $(1 - \beta) E_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$ both rise markedly as the households present value of income and borrowing rise pretty much together.

This example illustrates the following point: the definition of cointegration implies that the cointegrating residual is asymptotically covariance stationary, not covariance stationary.

The cointegrating residual for the specification with zero income and zero debt initially has a notable transient component that dominates its behavior early in the sample.

By altering initial conditions, we shall remove this transient in our second example to be presented below.
A borrowers and lenders closed economy

When we set $y_{-1} = y_{-2} = 0$ and $b_0 = 0$ in the preceding exercise, we make debt head north early in the sample.

Average debt in the cross-section rises and approaches asymptote.

We can regard these as outcomes of a small open economy that borrows from abroad at the fixed gross interest rate $R = r + 1$ in anticipation of rising incomes.

So with the economic primitives set as above, the economy converges to a steady state in which there is an excess aggregate supply of risk-free loans at a gross interest rate of $R$.

This excess supply is filled by foreigner lenders willing to make those loans.

We can use virtually the same code to rig a poor mans Bewley [Bew86] model in the following way:

- as before, we start everyone at $b_0 = 0$

- But instead of starting everyone at $y_{-1} = y_{-2} = 0$, we draw $\begin{bmatrix} y_{-1} \\ y_{-2} \end{bmatrix}$ from the invariant distribution of
the \( \{y_t\} \) process

This rigs a closed economy in which people are borrowing and lending with each other at a gross risk-free interest rate of \( R = \beta^{-1} \)

Across the group of people being analyzed, risk-free loans are in zero excess supply

We have arranged primitives so that \( R = \beta^{-1} \) clears the market for risk-free loans at zero aggregate excess supply

So the risk-free loans are being made from one person to another within our closed set of agent

There is no need for foreigners to lend to our group

Lets have a look at the corresponding figures

```julia
out = income_consumption_debt_series(A_LSS, C_LSS, G_LSS, mxbewley, sxbewley)
bsimb, csimb, ysimb = out[1:3]
cons_meanb, cons_varb, debt_meanb, debt_varb = out[4:end]

consumption_income_debt_figure(bsimb, csimb, ysimb)
```
The graphs confirm the following outcomes:

- As before, the consumption distribution spreads out over time
- But now there is some initial dispersion because there is *ex ante* heterogeneity in the initial draws of $\begin{bmatrix} y_{-1} \\ y_{-2} \end{bmatrix}$
  - As before, the cross-section distribution of debt spreads out over time
  - Unlike before, the average level of debt stays at zero, confirming that this is a closed borrower-and-lender economy
  - Now the cointegrating residual seems stationary, and not just asymptotically stationary

Let's have a look at the cointegration figure

```julia
cointegration_figure(bsimb, csimb)
```
3.14 Consumption and Tax Smoothing with Complete and Incomplete Markets

Contents

- Consumption and Tax Smoothing with Complete and Incomplete Markets
  - Overview
  - Background
  - Model 1 (Complete Markets)
  - Model 2 (One-Period Risk Free Debt Only)
  - Example: Tax Smoothing with Complete Markets
3.14.1 Overview

This lecture describes two types of consumption-smoothing and tax-smoothing models

- one is in the complete markets tradition of Lucas and Stokey [LS83]
- the other is in the incomplete markets tradition of Hall [Hal78] and Barro [Bar79]

Complete markets allow a consumer or government to buy or sell claims contingent on all possible states of the world

Incomplete markets allow a consumer or government to buy or sell only a limited set of securities, often only a single risk-free security

Hall [Hal78] and Barro [Bar79] both assumed that the only asset that can be traded is a risk-free one period bond

Hall assumed an exogenous stochastic process of nonfinancial income and an exogenous gross interest rate on one period risk-free debt that equals $\beta^{-1}$, where $\beta \in (0, 1)$ is also a consumers intertemporal discount factor

Barro [Bar79] made an analogous assumption about the risk-free interest rate in a tax-smoothing model that we regard as isomorphic to Halls consumption-smoothing model

We maintain Hall and Barros assumption about the interest rate when we describe an incomplete markets version of our model

In addition, we extend their assumption about the interest rate to an appropriate counterpart that we use in a complete markets model in the style of Lucas and Stokey [LS83]

While we are equally interested in consumption-smoothing and tax-smoothing models, for the most part we focus explicitly on consumption-smoothing versions of these models

But for each version of the consumption-smoothing model there is a natural tax-smoothing counterpart obtained simply by

- relabeling consumption as tax collections and nonfinancial income as government expenditures
- relabeling the consumers debt as the governments assets

For elaborations on this theme, please see Optimal Savings II: LQ Techniques and later parts of this lecture

Well consider two closely related alternative assumptions about the consumers exogenous nonfinancial income process (or in the tax-smoothing interpretation, the governments exogenous expenditure process):

- that it obeys a finite $N$ state Markov chain (setting $N = 2$ most of the time)
- that it is described by a linear state space model with a continuous state vector in $\mathbb{R}^n$ driven by a Gaussian vector iid shock process

Well spend most of this lecture studying the finite-state Markov specification, but will briefly treat the linear state space specification before concluding
Relationship to Other Lectures

This lecture can be viewed as a followup to *Optimal Savings II: LQ Techniques* and a warm up for a model of tax smoothing described in *Optimal Taxation with State-Contingent Debt*

Linear-quadratic versions of the Lucas-Stokey tax-smoothing model are described in *Optimal Taxation in an LQ Economy*

The key difference between those lectures and this one is

- Here the decision maker takes all prices as exogenous, meaning that his decisions do not affect them
- In *Optimal Taxation in an LQ Economy* and *Optimal Taxation with State-Contingent Debt*, the decision maker – the government in the case of these lectures – recognizes that his decisions affect prices

So these later lectures are partly about how the government should manipulate prices of government debt

3.14.2 Background

Outcomes in consumption-smoothing (or tax-smoothing) models emerge from two sources:

- a decision maker – a consumer in the consumption-smoothing model or a government in the tax-smoothing model – who wants to maximize an intertemporal objective function that expresses its preference for paths of consumption (or tax collections) that are smooth in the sense of not varying across time and Markov states
- a set of trading opportunities that allow the optimizer to transform a possibly erratic nonfinancial income (or government expenditure) process into a smoother consumption (or tax collections) process by purchasing or selling financial securities

In the complete markets version of the model, each period the consumer can buy or sell one-period ahead state-contingent securities whose payoffs depend on next periods realization of the Markov state

In the two-state Markov chain case, there are two such securities each period

In an $N$ state Markov state version of the model, $N$ such securities are traded each period

These state-contingent securities are commonly called Arrow securities, after Kenneth Arrow who first theorized about them

In the incomplete markets version of the model, the consumer can buy and sell only one security each period, a risk-free bond with gross return $\beta^{-1}$

Finite State Markov Income Process

In each version of the consumption-smoothing model, nonfinancial income is governed by a two-state Markov chain (its easy to generalize this to an $N$ state Markov chain)

In particular, the state of the world is given by $s_t$ that follows a Markov chain with transition probability matrix

$$P_{ij} = \mathbb{P}\{s_{t+1} = s_j \mid s_t = s_i\}$$
Nonfinancial income \( \{y_t\} \) obeys
\[
y_t = \begin{cases} 
\bar{y}_1 & \text{if } s_t = \bar{s}_1 \\
\bar{y}_2 & \text{if } s_t = \bar{s}_2
\end{cases}
\]

A consumer wishes to maximize
\[
\mathbb{E} \left[ \sum_{t=0}^{\infty} \beta^t u(c_t) \right] \text{ where } u(c_t) = -(c_t - \gamma)^2 \text{ and } 0 < \beta < 1
\]

Remark About Isomorphism

We can regard these as Barro [Bar79] tax-smoothing models if we set \( c_t = T_t \) and \( G_t = y_t \), where \( T_t \) is total tax collections and \( \{G_t\} \) is an exogenous government expenditures process.

Market Structure

The two models differ in how effectively the market structure allows the consumer to transfer resources across time and Markov states, there being more transfer opportunities in the complete markets setting than in the incomplete markets setting.

Watch how these differences in opportunities affect
- how smooth consumption is across time and Markov states
- how the consumer chooses to make his levels of indebtedness behave over time and across Markov states

3.14.3 Model 1 (Complete Markets)

At each date \( t \geq 0 \), the consumer trades one-period ahead Arrow securities.

We assume that prices of these securities are exogenous to the consumer (or in the tax-smoothing version of the model, to the government).

Exogenous means that they are unaffected by the decision maker.

In Markov state \( s_t \) at time \( t \), one unit of consumption in state \( s_{t+1} \) at time \( t+1 \) costs \( q(s_{t+1} | s_t) \) units of the time \( t \) consumption good.

At time \( t = 0 \), the consumer starts with an inherited level of debt due at time 0 of \( b_0 \) units of time 0 consumption goods.

The consumer's budget constraint at \( t \geq 0 \) in Markov state \( s_t \) is
\[
c_t + b_t \leq y(s_t) + \sum_j q(\tilde{s}_j | s_t) b_{t+1}(\tilde{s}_j | s_t)
\]
where \( b_t \) is the consumers one-period debt that falls due at time \( t \) and \( b_{t+1}(\bar{s}_j \mid s_t) \) are the consumers time \( t \) sales of the time \( t + 1 \) consumption good in Markov state \( \bar{s}_j \), a source of time \( t \) revenues.

An analogue of Hall’s assumption that the one-period risk-free gross interest rate is \( \beta^{-1} \) is

\[
q(\bar{s}_j \mid \bar{s}_i) = \beta P_{ij} \quad (3.126)
\]

To understand this, observe that in state \( \bar{s}_i \) it costs \( \sum_j q(\bar{s}_j \mid \bar{s}_i) \) to purchase one unit of consumption next period \textit{for sure}, i.e., meaning no matter what state of the world occurs at \( t + 1 \).

Hence the implied price of a risk-free claim on one unit of consumption next period is

\[
\sum_j q(\bar{s}_j \mid \bar{s}_i) = \sum_j \beta P_{ij} = \beta
\]

This confirms that (3.126) is a natural analogue of Hall’s assumption about the risk-free one-period interest rate.

First-order necessary conditions for maximizing the consumer’s expected utility are

\[
\beta \frac{u'(c_{t+1})}{u'(c_t)} P\{s_{t+1} \mid s_t\} = q(s_{t+1} \mid s_t)
\]

or, under our assumption (3.126) on Arrow security prices,

\[
c_{t+1} = c_t \quad (3.127)
\]

Thus, our consumer sets \( c_t = \bar{c} \) for all \( t \geq 0 \) for some value \( \bar{c} \) that it is our job now to determine.

**Guess:** Well make the plausible guess that

\[
b_{t+1}(\bar{s}_j \mid s_t = \bar{s}_i) = b(\bar{s}_j), \quad i = 1, 2; \; j = 1, 2 \quad (3.128)
\]

so that the amount borrowed today turns out to depend only on tomorrow’s Markov state. (Why is this a plausible guess?)

To determine \( \bar{c} \), we shall pursue the implications of the consumer’s budget constraints in each Markov state today and our guess (3.128) about the consumer’s debt level choices.

For \( t \geq 1 \), these imply

\[
\bar{c} + b(\bar{s}_1) = y(\bar{s}_1) + q(\bar{s}_1 \mid \bar{s}_1)b(\bar{s}_1) + q(\bar{s}_2 \mid \bar{s}_1)b(\bar{s}_2) \\
\bar{c} + b(\bar{s}_2) = y(\bar{s}_2) + q(\bar{s}_1 \mid \bar{s}_2)b(\bar{s}_1) + q(\bar{s}_2 \mid \bar{s}_2)b(\bar{s}_2),
\]

or

\[
\begin{bmatrix}
  b(\bar{s}_1) \\
  b(\bar{s}_2)
\end{bmatrix}
+ \begin{bmatrix}
  \bar{c} \\
  \bar{c}
\end{bmatrix}
= \begin{bmatrix}
  y(\bar{s}_1) \\
  y(\bar{s}_2)
\end{bmatrix}
+ \beta \begin{bmatrix}
  P_{11} & P_{12} \\
  P_{21} & P_{22}
\end{bmatrix}
\begin{bmatrix}
  b(\bar{s}_1) \\
  b(\bar{s}_2)
\end{bmatrix}
\]

**Chapter 3. Dynamic Programming**
These are 2 equations in the 3 unknowns \( \bar{c}, b(\bar{s}_1), b(\bar{s}_2) \)

To get a third equation, we assume that at time \( t = 0 \), \( b_0 \) is the debt due; and we assume that at time \( t = 0 \), the Markov state is \( \bar{s}_1 \)

Then the budget constraint at time \( t = 0 \) is

\[
\bar{c} + b_0 = y(\bar{s}_1) + q(\bar{s}_1 \mid \bar{s}_1)b(\bar{s}_2) + q(\bar{s}_2 \mid \bar{s}_1)b(\bar{s}_2)
\]

(3.130)

If we substitute (3.130) into the first equation of (3.129) and rearrange, we discover that

\[
b(\bar{s}_1) = b_0
\]

(3.131)

We can then use the second equation of (3.129) to deduce the restriction

\[
y(\bar{s}_1) - y(\bar{s}_2) + [q(\bar{s}_1 \mid \bar{s}_1) - q(\bar{s}_1 \mid \bar{s}_2) - 1]b_0 + [q(\bar{s}_2 \mid \bar{s}_1) + 1 - q(\bar{s}_2 \mid \bar{s}_2)]b(\bar{s}_2) = 0,
\]

(3.132)

an equation in the unknown \( b(\bar{s}_2) \)

Knowing \( b(\bar{s}_1) \) and \( b(\bar{s}_2) \), we can solve equation (3.130) for the constant level of consumption \( \bar{c} \)

**Key outcomes**

The preceding calculations indicate that in the complete markets version of our model, we obtain the following striking results:

- The consumer chooses to make consumption perfectly constant across time and Markov states

We computed the constant level of consumption \( \bar{c} \) and indicated how that level depends on the underlying specifications of preferences, Arrow securities prices, the stochastic process of exogenous nonfinancial income, and the initial debt level \( b_0 \)

- The consumer’s debt neither accumulates, nor decumulates, nor drifts. Instead the debt level each period is an exact function of the Markov state, so in the two-state Markov case, it switches between two values

- We have verified guess (3.128)

We computed how one of those debt levels depends entirely on initial debt – it equals it – and how the other value depends on virtually all remaining parameters of the model

**Code**

Here is some code that, among other things, contains a function called `consumption_complete()`

This function computes \( b(\bar{s}_1), b(\bar{s}_2), \bar{c} \) as outcomes given a set of parameters, under the assumption of complete markets
using QuantEcon

""
The data for a consumption problem, including some default values.
""
struct ConsumptionProblem{TF <: AbstractFloat}
    β::TF
    y::Vector{TF}
    b0::TF
    P::Matrix{TF}
end

""
Parameters

---

β : discount factor
P : 2x2 transition matrix
y : Array containing the two income levels
b0 : debt in period 0 (= state_1 debt level)

""
function ConsumptionProblem(β = 0.96, 
    y = [2.0, 1.5],
    b0 = 3.0,
    P = [0.8 0.2;
         0.4 0.6])
    ConsumptionProblem(β, y, b0, P)
end

""
Computes endogenous values for the complete market case.

Parameters

---

cp : instance of ConsumptionProblem

Returns

---

c_bar : constant consumption
b1 : rolled over b0
b2 : debt in state_2

associated with the price system

    Q = β * P

""
function consumption_complete(cp::ConsumptionProblem)
\( \beta, P, y, b0 = cp.\beta, cp.P, cp.y, cp.b0 \)  # Unpack

\( y1, y2 = y \)  # extract income levels

\( b1 = b0 \)  # b1 is known to be equal to b0

\( Q = \beta \times P \)  # assumed price system

# Using equation (7) calculate b2

\( b2 = (y2 - y1 - (Q[1, 1] - Q[2, 1] - 1) \times b1) / (Q[1, 2] + 1 - Q[2, 2]) \)

# Using equation (5) calculate c_bar

\( c_{\text{bar}} = y1 - b0 + ([b1 b2] \times Q[1, :)])[1] \)

return \( c_{\text{bar}}, b1, b2 \)

end

""
Computes endogenous values for the incomplete market case.

Parameters
----------

\( cp \): instance of ConsumptionProblem

\( N_{\text{simul}} \): Integer

""

function consumption_incomplete(cp::ConsumptionProblem;
                              N_simul::Integer=150)

\( \beta, P, y, b0 = cp.\beta, cp.P, cp.y, cp.b0 \)  # Unpack

# For the simulation use the MarkovChain type

mc = MarkovChain(P)

# Useful variables

\( y = y' \)

\( v = \text{inv}(\text{eye}(2) - \beta \times P) \times y \)

# Simulate state path

s_path = simulate(mc, N_simul, init=1)

# Store consumption and debt path

b_path, c_path = ones(N_simul + 1), ones(N_simul)
b_path[1] = b0

# Optimal decisions from (12) and (13)

db = ((1 - \beta) \times v - y) / \beta

for \( (i, s) \) in enumerate(s_path)
    c_path[i] = (1 - \beta) \times (v - b_path[i] \times \text{ones}(2, 1))[s, 1]
    b_path[i + 1] = b_path[i] + db[s, 1]
end

return c_path, b_path[1:end-1], y[s_path], s_path
end

3.14. Consumption and Tax Smoothing with Complete and Incomplete Markets 521
Lets test by checking that \( c \) and \( b_2 \) satisfy the budget constraint

```julia
cp = ConsumptionProblem()
c_bar, b1, b2 = consumption_complete(cp)
debt_complete = [b1, b2]
isapprox((c_bar + b2 - cp.y[2] - debt_complete' * (cp.β * cp.P)[2, :])[1], 0)
```

true

Below, well take the outcomes produced by this code – in particular the implied consumption and debt paths – and compare them with outcomes from an incomplete markets model in the spirit of Hall [Hal78] and Barro [Bar79] (and also, for those who love history, Gallatin (1807) [Gal37])

**3.14.4 Model 2 (One-Period Risk Free Debt Only)**

This is a version of the original models of Hall (1978) and Barro (1979) in which the decision makers ability to substitute intertemporally is constrained by his ability to buy or sell only one security, a risk-free one-period bond bearing a constant gross interest rate that equals \( \beta^{-1} \)

Given an initial debt \( b_0 \) at time 0, the consumer faces a sequence of budget constraints

\[
c_t + b_t = y_t + \beta b_{t+1}, \quad t \geq 0
\]

where \( \beta \) is the price at time \( t \) of a risk-free claim on one unit of time consumption at time \( t + 1 \)

First-order conditions for the consumers problem are

\[
\sum_j u'(c_{t+1,j})P_{ij} = u'(c_{t,i})
\]

For our assumed quadratic utility function this implies

\[
\sum_j c_{t+1,j}P_{ij} = c_{t,i}, \quad (3.133)
\]

which is Halls (1978) conclusion that consumption follows a random walk

As we saw in our first lecture on the *permanent income model*, this leads to

\[
b_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - (1 - \beta)^{-1} c_t \quad (3.134)
\]

and

\[
c_t = (1 - \beta) \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - b_t \quad . \quad (3.135)
\]
Equation (3.135) expresses $c_t$ as a net interest rate factor $1 - \beta$ times the sum of the expected present value of nonfinancial income $E_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$ and financial wealth $-b_t$

Substituting (3.135) into the one-period budget constraint and rearranging leads to

$$b_{t+1} - b_t = \beta^{-1} \left[ (1 - \beta)E_t \sum_{j=0}^{\infty} \beta^j y_{t+j} - y_t \right]$$

(3.136)

Now lets do a useful calculation that will yield a convenient expression for the key term $E_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$ in our finite Markov chain setting

Define

$$v_t := E_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$$

In our finite Markov chain setting, $v_t = v(1)$ when $s_t = s_1$ and $v_t = v(2)$ when $s_t = s_2$

Therefore, we can write

$$v(1) = y(1) + \beta P_{11} v(1) + \beta P_{12} v(2)$$
$$v(2) = y(2) + \beta P_{21} v(1) + \beta P_{22} v(2)$$

or

$$\vec{v} = \vec{y} + \beta P \vec{v}$$

where $\vec{v} = \begin{bmatrix} v(1) \\ v(2) \end{bmatrix}$ and $\vec{y} = \begin{bmatrix} y(1) \\ y(2) \end{bmatrix}$

We can also write the last expression as

$$\vec{v} = (I - \beta P)^{-1} \vec{y}$$

In our finite Markov chain setting, from expression (3.135), consumption at date $t$ when debt is $b_t$ and the Markov state today is $s_t = i$ is evidently

$$c(b_t, i) = (1 - \beta) \left( [(I - \beta P)^{-1} \vec{y}]_i - b_t \right)$$

(3.137)

and the increment in debt is

$$b_{t+1} - b_t = \beta^{-1} \left[ (1 - \beta)v(i) - y(i) \right]$$

(3.138)
Summary of Outcomes

In contrast to outcomes in the complete markets model, in the incomplete markets model

- consumption drifts over time as a random walk; the level of consumption at time $t$ depends on the level of debt that the consumer brings into the period as well as the expected discounted present value of nonfinancial income at $t$
- the consumers debt drifts upward over time in response to low realizations of nonfinancial income and drifts downward over time in response to high realizations of nonfinancial income
- the drift over time in the consumers debt and the dependence of current consumption on todays debt level account for the drift over time in consumption

The Incomplete Markets Model

The code above also contains a function called $\text{consumption\_incomplete()}$ that uses (3.137) and (3.138) to

- simulate paths of $y_t, c_t, b_{t+1}$
- plot these against values of of $\bar{c}, b(s_1), b(s_2)$ found in a corresponding complete markets economy

Let's try this, using the same parameters in both complete and incomplete markets economies

```python
using PyPlot
srand(1)
N_simul = 150
cp = ConsumptionProblem()
c_bar, b1, b2 = consumption_complete(cp)
debt_complete = [b1, b2]
c_path, debt_path, y_path, s_path = consumption_incomplete(cp, N_simul=N_simul)
fig, ax = subplots(1, 2, figsize=(15, 5))
ax[1][:set_title]("Consumption paths")
ax[1][:plot](1:N_simul, c_path, label="incomplete market")
ax[1][:plot](1:N_simul, c_bar * ones(N_simul), label="complete market")
ax[1][:plot](1:N_simul, y_path, label="income", lw=2, alpha=.6, ls="--")
ax[1][:legend]()
ax[1][:set_xlabel]("Periods")
ax[1][:set_xlim](1.4, 2.1)

ax[2][:set_title]("Debt paths")
ax[2][:plot](1:N_simul, debt_path, label="incomplete market")
ax[2][:plot](1:N_simul, debt_complete[s_path], label="complete market")
ax[2][:plot](1:N_simul, y_path, label="income", alpha=.6, ls="--")
ax[2][:legend]()
```
In the graph on the left, for the same sample path of nonfinancial income $y_t$, notice that

- consumption is constant when there are complete markets, but it takes a random walk in the incomplete markets version of the model
- the consumers debt oscillates between two values that are functions of the Markov state in the complete markets model, while the consumers debt drifts in a unit root fashion in the incomplete markets economy

**Using the Isomorphism**

We can simply relabel variables to acquire tax-smoothing interpretations of our two models.
Example: Tax Smoothing with Complete Markets

It is useful to focus on a simple tax-smoothing example with complete markets.

This example will illustrate how, in a complete markets model like that of Lucas and Stokey \cite{LS83}, the government purchases insurance from the private sector.

- Purchasing insurance protects the government against the need to raise taxes too high or issue too much debt in the high government expenditure event.

We assume that government expenditures move between two values $G_1 < G_2$, where Markov state 1 means peace and Markov state 2 means war.

The government budget constraint in Markov state $i$ is

$$T_i + b_i = G_i + \sum_j Q_{ij} b_j$$

where

$$Q_{ij} = \beta P_{ij}$$

is the price of one unit of output next period in state $j$ when today’s Markov state is $i$ and $b_i$ is the government’s level of assets in Markov state $i$.

That is, $b_i$ is the amount of the one-period loans owned by the government that fall due at time $t$.

As above, we assume that the initial Markov state is state 1.

In addition, to simplify our example, we will set the government’s initial asset level to 0, so that $b_1 = 0$.

Here is our code to compute a quantitative example with zero debt in peace time:

```julia
# Parameters
\beta = 0.96
y = [1.0, 2.0]
b0 = 0.0
```
\[ P = [0.8, 0.2; \\
    0.4, 0.6] \]

\[ \text{cp = ConsumptionProblem(β, y, b0, P)} \]

\[ Q = \beta \cdot P \]

\[ \text{N_simul = 150} \]

\[ \text{c_bar, b1, b2 = consumption_complete(cp)} \]

\[ \text{debt_complete = [b1, b2]} \]

\[ \text{println("P = \$P")} \]

\[ \text{println("Q = \$Q")} \]

\[ \text{println("Govt expenditures in peace and war = \$y")} \]

\[ \text{println("Constant tax collections = \$c_bar")} \]

\[ \text{println("Govt assets in two states = \$debt_complete")} \]

\[ \text{msg = ""} \]

\[ \text{println(msg)} \]

\[ \text{AS1 = Q[1, 2] * b2} \]

\[ \text{println("Spending on Arrow war security in peace = \$AS1")} \]

\[ \text{AS2 = Q[2, 2] * b2} \]

\[ \text{println("Spending on Arrow war security in war = \$AS2")} \]

\[ \text{println("\n")} \]

\[ \text{println("Government tax collections plus asset levels in peace and war")} \]

\[ \text{TB1 = c_bar + b1} \]

\[ \text{println("T+b in peace = \$TB1")} \]

\[ \text{TB2 = c_bar + b2} \]

\[ \text{println("T+b in war = \$TB2")} \]

\[ \text{println("\n")} \]

\[ \text{println("Total government spending in peace and war")} \]

\[ \text{G1= y[1] + AS1} \]

\[ \text{G2 = y[2] + AS2} \]

\[ \text{println("total govt spending in peace = \$G1")} \]

\[ \text{println("total govt spending in war = \$G2")} \]

\[ \text{println("\n")} \]

\[ \text{println("Let's see ex post and ex ante returns on Arrow securities")} \]

\[ \Pi = 1 ./ Q \quad \# \text{ reciprocal(Q)} \]

\[ \text{exret = Π} \]

\[ \text{println("Ex post returns to purchase of Arrow securities = \$exret")} \]

\[ \text{exant = Π \cdot P} \]

\[ \text{println("Ex ante returns to purchase of Arrow securities = \$exant")} \]
\[
\begin{align*}
P &= \begin{bmatrix} 0.8 & 0.2 \\ 0.4 & 0.6 \end{bmatrix} \\
Q &= \begin{bmatrix} 0.768 & 0.192 \\ 0.384 & 0.576 \end{bmatrix} \\
\text{Govt expenditures in peace and war} &= [1.0, 2.0] \\
\text{Constant tax collections} &= 1.3116883116883118 \\
\text{Govt assets in two states} &= [0.0, 1.62338] \\
\text{Now let's check the government's budget constraint in peace and war. Our assumptions imply that the government always purchases 0 units of the Arrow peace security.} \\
\text{Spending on Arrow war security in peace} &= 0.3116883116883117 \\
\text{Spending on Arrow war security in war} &= 0.9350649350649349 \\
\text{Government tax collections plus asset levels in peace and war} \\
T+b \text{ in peace} &= 1.3116883116883118 \\
T+b \text{ in war} &= 2.9350649350649354 \\
\text{Total government spending in peace and war} \\
\text{total govt spending in peace} &= 1.3116883116883118 \\
\text{total govt spending in war} &= 2.935064935064935 \\
\text{Let's see ex post and ex ante returns on Arrow securities} \\
\text{Ex post returns to purchase of Arrow securities} &= [1.30208, 5.20833, 2.60417, 1.73611] \\
\text{Ex ante returns to purchase of Arrow securities} &= [1.04167, 1.04167, 1.04167, 1.04167] \\
\end{align*}
\]

**Explanation**

In this example, the government always purchase 0 units of the Arrow security that pays off in peace time (Markov state 1)

But it purchases a positive amount of the security that pays off in war time (Markov state 2)

We recommend plugging the quantities computed above into the government budget constraints in the two Markov states and staring

This is an example in which the government purchases *insurance* against the possibility that war breaks out or continues

- the insurance does not pay off so long as peace continues
- the insurance pays off when there is war

*Exercise*: try changing the Markov transition matrix so that

\[
P = \begin{bmatrix} 1 & 0 \\ 0.2 & 0.8 \end{bmatrix}
\]

Also, start the system in Markov state 2 (war) with initial government assets \(-10\), so that the government starts the war in debt and \(b_2 = -10\)
3.14.6 Linear State Space Version of Complete Markets Model

Now we use a setting like that in first lecture on the permanent income model

In that model, there were

- incomplete markets: the consumer could trade only a single risk-free one-period bond bearing gross one-period risk-free interest rate equal to $\beta^{-1}$
- the consumers exogenous nonfinancial income was governed by a linear state space model driven by Gaussian shocks, the kind of model studied in an earlier lecture about linear state space models

Well write down a complete markets counterpart of that model

So now well suppose that nonfinancial income is governed by the state space system

$$
x_{t+1} = Ax_t + Cw_{t+1}

y_t = S_g x_t
$$

where $x_t$ is an $n \times 1$ vector and $w_{t+1} \sim N(0, I)$ is IID over time

Again, as a counterpart of the Hall-Barro assumption that the risk-free gross interest rate is $\beta^{-1}$, we assume the scaled prices of one-period ahead Arrow securities are

$$
p_{t+1}(x_{t+1} | x_t) = \beta\phi(x_{t+1} | Ax_t, CC')
$$

(3.139)

where $\phi(\cdot | \mu, \Sigma)$ is a multivariate Gaussian distribution with mean vector $\mu$ and covariance matrix $\Sigma$

Let $b(x_{t+1})$ be a vector of state-contingent debt due at $t + 1$ as a function of the $t + 1$ state $x_{t+1}$.

Using the pricing function assumed in (3.139), the value at $t$ of $b(x_{t+1})$ is

$$
\beta \int b(x_{t+1})\phi(x_{t+1} | Ax_t, CC')dx_{t+1} = \beta\mathbb{E}_t b_{t+1}
$$

In the complete markets setting, the consumer faces a sequence of budget constraints

$$
c_t + b_t = y_t + \beta\mathbb{E}_t b_{t+1}, t \geq 0
$$

We can solve the time $t$ budget constraint forward to obtain

$$
b_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j (y_{t+j} - c_{t+j})
$$

We assume as before that the consumer cares about the expected value of

$$
\sum_{t=0}^{\infty} \beta^t u(c_t), \quad 0 < \beta < 1
$$

In the incomplete markets version of the model, we assumed that $u(c_t) = -(c_t - \gamma)^2$, so that the above utility functional became

$$
-\sum_{t=0}^{\infty} \beta^t (c_t - \gamma)^2, \quad 0 < \beta < 1
$$
But in the complete markets version, we can assume a more general form of utility function that satisfies $u' > 0$ and $u'' < 0$

The first-order condition for the consumers problem with complete markets and our assumption about Arrow securities prices is

$$u'(c_{t+1}) = u'(c_t) \quad \text{for all } t \geq 0$$

which again implies $c_t = \bar{c}$ for some $\bar{c}$

So it follows that

$$b_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j (y_{t+j} - \bar{c})$$

or

$$b_t = S_y(I - \beta A)^{-1} x_t - \frac{1}{1 - \beta} \bar{c}$$

(3.140)

where the value of $\bar{c}$ satisfies

$$\bar{b}_0 = S_y(I - \beta A)^{-1} x_0 - \frac{1}{1 - \beta} \bar{c}$$

(3.141)

where $\bar{b}_0$ is an initial level of the consumers debt, specified as a parameter of the problem

Thus, in the complete markets version of the consumption-smoothing model, $c_t = \bar{c}, \forall t \geq 0$ is determined by (3.141) and the consumers debt is a fixed function of the state $x_t$ described by (3.140)

Here's an example that shows how in this setting the availability of insurance against fluctuating nonfinancial income allows the consumer completely to smooth consumption across time and across states of the world.

---

```julia
# Computes the path of consumption and debt for the previously described complete markets model where exogenous income follows a linear state space

function complete_ss(β::AbstractFloat,
    b0::Union{AbstractFloat, Array},
    x0::Union{AbstractFloat, Array},
    A::Union{AbstractFloat, Array},
    C::Union{AbstractFloat, Array},
    S_y::Union{AbstractFloat, Array},
    T::Integer=12)

    # Create a linear state space for simulation purposes
    # This adds "b" as a state to the linear state space system
    # so that setting the seed places shocks in same place for
    # both the complete and incomplete markets economy
    # Atilde = vcat(hcat(A, zeros(size(A,1), 1)),
    #    zeros(1, size(A,2) + 1))
```

---
```julia
# C_tilde = vcat(C, zeros(1, 1))
# S_ytilde = hcat(S_y, zeros(1, 1))

lss = LSS(A, C, S_y, mu_0=x0)

# Add extra state to initial condition
# x0 = hcat(x0, 0)
# Compute the (I - β*A)^{-1}
rm = inv(eye(size(A, 1)) - β * A)

# Constant level of consumption
 lineno = (1 - β) * (S_y * rm * x0 - b0)
c_hist = ones(T) * lineno

# Debt
 x_hist, y_hist = simulate(lss, T)
b_hist = (S_y * rm * x_hist - lineno) / (1.0 - β)

return c_hist, vec(b_hist), vec(y_hist), x_hist

N_simul = 150

# Define parameters
α = 1.0
# N_simul = 1
# T = N_simul
A = [1.0 0.0 0.0;
     ρ1 ρ1 ρ2;
     0.0 1.0 0.0]
C = [0.0, α, 0.0]
S_y = [1.0 1.0 0.0]
β, b0 = 0.95, -10.0
x0 = [1.0, α / (1 - ρ1), α / (1 - ρ1)]

# Do simulation for complete markets
s = rand(1:10000)
srand(s)  # Seeds get set the same for both economies
out = complete_ss(β, b0, x0, A, C, S_y, 150)
c_hist_com, b_hist_com, y_hist_com, x_hist_com = out

fig, ax = subplots(1, 2, figsize=(15, 5))

# Consumption plots
ax[1][:set_title]("Cons and income")
ax[1][:plot](1:N_simul, c_hist_com, label="consumption")
ax[1][:plot](1:N_simul, y_hist_com, label="income",
            lw=2, alpha=0.6, ls="--")
ax[1][:legend]()
```

3.14. Consumption and Tax Smoothing with Complete and Incomplete Markets
Interpretation of Graph

In the above graph, please note that:

- nonfinancial income fluctuates in a stationary manner
- consumption is completely constant
- the consumers’ debt fluctuates in a stationary manner; in fact, in this case because nonfinancial income is a first-order autoregressive process, the consumers’ debt is an exact affine function (meaning linear plus a constant) of the consumers’ nonfinancial income

Incomplete Markets Version

The incomplete markets version of the model with nonfinancial income being governed by a linear state space system is described in the first lecture on the permanent income model and the followup lecture on the permanent income model.

In that version, consumption follows a random walk and the consumers’ debt follows a process with a unit root.

We leave it to the reader to apply the usual isomorphism to deduce the corresponding implications for a tax-smoothing model like Barros [Bar79].
Government Manipulation of Arrow Securities Prices

In *optimal taxation in an LQ economy* and *recursive optimal taxation*, we study *complete-markets* models in which the government recognizes that it can manipulate Arrow securities prices.

In *optimal taxation with incomplete markets*, we study an *incomplete-markets* model in which the government manipulates asset prices.

### 3.15 Optimal Savings III: Occasionally Binding Constraints

#### Contents

- Optimal Savings III: Occasionally Binding Constraints
  - Overview
  - The Optimal Savings Problem
  - Computation
  - Exercises
  - Solutions

#### 3.15.1 Overview

Next we study an optimal savings problem for an infinitely lived consumer—the common ancestor described in [LS18], section 1.3.

This is an essential sub-problem for many representative macroeconomic models:

- [Aiy94]
- [Hug93]
- etc.

It is related to the decision problem in the *stochastic optimal growth model* and yet differs in important ways.

For example, the choice problem for the agent includes an additive income term that leads to an occasionally binding constraint.

Our presentation of the model will be relatively brief:

- For further details on economic intuition, implication and models, see [LS18]
- Proofs of all mathematical results stated below can be found in this paper

To solve the model we will use Euler equation based time iteration, similar to *this lecture*.

This method turns out to be:

- Globally convergent under mild assumptions, even when utility is unbounded (both above and below)
• More efficient numerically than value function iteration

References

Other useful references include [Dea91], [DH10], [Kuh13], [Rab02], [Rei09] and [SE77]

3.15.2 The Optimal Savings Problem

Let's write down the model and then discuss how to solve it

Set Up

Consider a household that chooses a state-contingent consumption plan \( \{c_t\}_{t \geq 0} \) to maximize

\[
\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)
\]

subject to

\[
c_t + a_{t+1} \leq R a_t + z_t, \quad c_t \geq 0, \quad a_t \geq -b \quad t = 0, 1, \ldots
\]

Here

• \( \beta \in (0, 1) \) is the discount factor
• \( a_t \) is asset holdings at time \( t \), with ad-hoc borrowing constraint \( a_t \geq -b \)
• \( c_t \) is consumption
• \( z_t \) is non-capital income (wages, unemployment compensation, etc.)
• \( R := 1 + r \), where \( r > 0 \) is the interest rate on savings

Non-capital income \( \{z_t\} \) is assumed to be a Markov process taking values in \( Z \subset (0, \infty) \) with stochastic kernel \( \Pi \)

This means that \( \Pi(z, B) \) is the probability that \( z_{t+1} \in B \) given \( z_t = z \)

The expectation of \( f(z_{t+1}) \) given \( z_t = z \) is written as

\[
\int f(\hat{z}) \Pi(z, d\hat{z})
\]

We further assume that

1. \( r > 0 \) and \( \beta R < 1 \)
2. \( u \) is smooth, strictly increasing and strictly concave with \( \lim_{c \to 0} u'(c) = \infty \) and \( \lim_{c \to \infty} u'(c) = 0 \)
The asset space is \([-b, \infty)\) and the state is the pair \((a, z) \in S := [-b, \infty) \times Z\)

A feasible consumption path from \((a, z) \in S\) is a consumption sequence \(\{c_t\}\) such that \(\{c_t\}\) and its induced asset path \(\{a_t\}\) satisfy

1. \((a_0, z_0) = (a, z)\)
2. the feasibility constraints in (3.142), and
3. measurability of \(c_t\) w.r.t. the filtration generated by \(\{z_1, \ldots, z_t\}\)

The meaning of the third point is just that consumption at time \(t\) can only be a function of outcomes that have already been observed.

**Value Function and Euler Equation**

The value function \(V : S \to \mathbb{R}\) is defined by

\[
V(a, z) := \sup \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\}
\]

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 (3.143)

To pin down such paths we can use a version of the Euler equation, which in the present setting is

\[
u'(c_t) \geq \beta R \mathbb{E}_t[u'(c_{t+1})]
\]

and

\[
u'(c_t) = \beta R \mathbb{E}_t[u'(c_{t+1})] \quad \text{whenever } c_t < Ra_t + z_t + b
\]

In essence, this says that the natural arbitrage relation \(u'(c_t) = \beta R \mathbb{E}_t[u'(c_{t+1})]\) holds when the choice of current consumption is interior

Interiority means that \(c_t\) is strictly less than its upper bound \(Ra_t + z_t + b\)

(The lower boundary case \(c_t = 0\) never arises at the optimum because \(u'(0) = \infty\) )

When \(c_t\) does hit the upper bound \(Ra_t + z_t + b\), the strict inequality \(u'(c_t) > \beta R \mathbb{E}_t[u'(c_{t+1})]\) can occur because \(c_t\) cannot increase sufficiently to attain equality

With some thought and effort, one can show that (3.144) and (3.145) are equivalent to

\[
u'(c_t) = \max \{\beta R \mathbb{E}_t[u'(c_{t+1})], u'(Ra_t + z_t + b)\}
\]
Optimality Results

Given our assumptions, it is known that

1. For each \((a, z) \in S\), a unique optimal consumption path from \((a, z)\) exists

2. This path is the unique feasible path from \((a, z)\) satisfying the Euler equality (3.146) and the transversality condition

\[
\lim_{t \to \infty} \beta t \mathbb{E}\left[u'(c_t)a_{t+1}\right] = 0. \tag{3.147}
\]

Moreover, there exists an optimal consumption function \(c^*: S \to [0, \infty)\) such that the path from \((a, z)\) generated by

\[
(a_0, z_0) = (a, z), \quad z_{t+1} \sim \Pi(z_t, dy), \quad c_t = c^*(a_t, z_t) \quad \text{and} \quad a_{t+1} = Ra_t + z_t - c_t
\]

satisfies both (3.146) and (3.147), and hence is the unique optimal path from \((a, z)\)

In summary, to solve the optimization problem, we need to compute \(c^*\)

### 3.15.3 Computation

There are two standard ways to solve for \(c^*\)

1. Time iteration (TI) using the Euler equality
2. Value function iteration (VFI)

Let's look at these in turn

#### Time Iteration

We can rewrite (3.146) to make it a statement about functions rather than random variables

In particular, consider the functional equation

\[
u' \circ c(a, z) = \max \left\{ \gamma \int u' \circ c \{Ra + z - c(a, z), \, z\} \Pi(z, dz), \, u'(Ra + z + b) \right\} \tag{3.148}\]

where \(\gamma := \beta R\) and \(u' \circ c(s) := u'(c(s))\)

Equation (3.148) is a functional equation in \(c\)

In order to identify a solution, let \(\mathcal{C}\) be the set of candidate consumption functions \(c: S \to \mathbb{R}\) such that

- each \(c \in \mathcal{C}\) is continuous and (weakly) increasing
- \(\min Z \leq c(a, z) \leq Ra + z + b\) for all \((a, z) \in S\)

In addition, let \(K: \mathcal{C} \to \mathcal{C}\) be defined as follows:

For given \(c \in \mathcal{C}\), the value \(Kc(a, z)\) is the unique \(t \in J(a, z)\) that solves
\[ u'(t) = \max \left\{ \gamma \int u' \circ c \{ Ra + z - t, \dot{z} \} \Pi(z, d\dot{z}), u'(Ra + z + b) \right\} \]  

(3.149)

where

\[ J(a, z) := \{ t \in \mathbb{R} : \min Z \leq t \leq Ra + z + b \} \]  

(3.150)

We refer to \( K \) as Colemans policy function operator \([\text{Col90}]\).

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 \( K^n c \to c^* \) as \( n \to \infty \) for any \( c \in \mathcal{C} \).

By the definition of \( K \), the fixed points of \( K \) in \( \mathcal{C} \) coincide with the solutions to (3.148) in \( \mathcal{C} \).

In particular, it can be shown that the path \( \{ c_t \} \) generated from \( (a_0, z_0) \in S \) using policy function \( c^* \) is the unique optimal path from \( (a_0, z_0) \in S \).

**TLDR** The unique optimal policy can be computed by picking any \( c \in \mathcal{C} \) and iterating with the operator \( K \) defined in (3.149).

### Value Function Iteration

The Bellman operator for this problem is given by

\[ T v(a, z) = \max_{0 \leq c \leq Ra + z + b} \left\{ u(c) + \beta \int v(Ra + z - c, \dot{z}) \Pi(z, d\dot{z}) \right\} \]  

(3.151)

We have to be careful with VFI (i.e., iterating with \( T \)) in this setting because \( u \) is not assumed to be bounded

- In fact typically unbounded both above and below e.g. \( u(c) = \log c \)
- In which case, the standard DP theory does not apply
- \( T^n v \) is not guaranteed to converge to the value function for arbitrary continuous bounded \( v \)

Nonetheless, we can always try the popular strategy iterate and hope

We can then check the outcome by comparing with that produced by TI

The latter is known to converge, as described above

---

3.15. Optimal Savings III: Occasionally Binding Constraints
Implementation

Here's the code for a type called `ConsumerProblem` that stores primitives, as well as

- a `bellman_operator` function, which implements the Bellman operator $T$ specified above
- a `coleman_operator` function, which implements the Coleman operator $K$ specified above
- an `initialize` function, which generates suitable initial conditions for iteration

```julia
using QuantEcon
using Optim

# utility and marginal utility functions
u(x) = log(x)
ku(x) = 1 / x

""
Income fluctuation problem
""

#### Fields
- `r::Real` : Strictly positive interest rate
- `R::Real` : The interest rate plus 1 (strictly greater than 1)
- `β::Real` : Discount rate in $(0, 1)$
- `b::Real` : The borrowing constraint
- `Π::Matrix{T}` where $T::Real` : Transition matrix for `z`
- `z_vals::Vector{T}` where $T::Real` : Levels of productivity
- `asset_grid::AbstractArray` : Grid of asset values

""
struct ConsumerProblem{T <: Real}
    r::T
    R::T
    β::T
    b::T
    Π::Matrix{T}
    z_vals::Vector{T}
    asset_grid::AbstractArray
end

function ConsumerProblem(;r=0.01,
                          β=0.96,
                          Π=[0.6 0.4; 0.05 0.95],
                          z_vals=[0.5, 1.0],
                          b=0.0,
                          grid_max=16,
                          grid_size=50)
    R = 1 + r
    asset_grid = linspace(-b, grid_max, grid_size)
    ConsumerProblem(r, R, β, b, Π, z_vals, asset_grid)
end
```
Apply the Bellman operator for a given model and initial value.

#### 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`.

```julia
function bellman_operator!(cp::ConsumerProblem,
    V::Matrix,
    out::Matrix;
    ret_policy::Bool=false)

    # simplify names, set up arrays
    R, Π, β, b = cp.R, cp.Π, cp.β, cp.b
    asset_grid, z_vals = cp.asset_grid, cp.z_vals
    z_idx = 1:length(z_vals)

    # value function when the shock index is z_i
    vf = interp(asset_grid, V)
    opt_lb = 1e-8

    # 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)
                EV = dot(vf.(R .+ a .+ z .- c, z_idx), Π[i_z, :]) # compute expectation
                return -u(c) - β * EV
            end
            res = optimize(obj, opt_lb, R .* a .+ z .+ b)
            c_star = Optim.minimizer(res)

            if ret_policy
                out[i_a, i_z] = c_star
            else
                out[i_a, i_z] = - Optim.minimum(res)
            end
        end
    end
end
```
bellman_operator(cp::ConsumerProblem, V::Matrix; ret_policy=false) =
    bellman_operator!(cp, V, similar(V); ret_policy=ret_policy)

""
Extract the greedy policy (policy function) of the model.

##### 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

""
greedy!(cp::ConsumerProblem, V::Matrix, out::Matrix) =
    update_bellman!(cp, V, out, ret_policy=true)

greedy(cp::ConsumerProblem, V::Matrix) =
    update_bellman(cp, V, ret_policy=true)

""
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 of 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
    asset_grid, z_vals = cp.asset_grid, cp.z_vals
    z_idx = 1:length(z_vals)
    gam = R * β

# policy function when the shock index is $z_i$

```julia
cf = interp(asset_grid, c)
```

# compute lower_bound for optimization

```julia
opt_lb = 1e-8
```

```julia
for (i_z, z) in enumerate(z_vals)
    for (i_a, a) in enumerate(asset_grid)
        function h(t)
            cps = cf.(R * a + z - t, z_idx)  # $c'$ for each $z'$
            expectation = dot(du.(cps), Π[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=Optim.Brent())
        out[i_a, i_z] = Optim.minimizer(res)
    end
end
```

```julia
# simplifies names, set up arrays
R, β, b = cp.R, cp.β, 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...)
```

```julia
# 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 - β)
    end
end
```

Both `bellman_operator` and `coleman_operator` use linear interpolation along the asset grid to approximate the value and consumption functions

3.15. Optimal Savings III: Occasionally Binding Constraints
The following exercises walk you through several applications where policy functions are computed.

In exercise 1 you will see that while VFI and TI produce similar results, the latter is much faster.

Intuition behind this fact was provided in a previous lecture on time iteration.

### 3.15.4 Exercises

#### Exercise 1

The first exercise is to replicate the following figure, which compares TI and VFI as solution methods.

The figure shows consumption policies computed by iteration of $K$ and $T$ respectively.

- In the case of iteration with $T$, the final value function is used to compute the observed policy.

Consumption is shown as a function of assets with income $z$ held fixed at its smallest value.

The following details are needed to replicate the figure:

- The parameters are the default parameters in the definition of `consumerProblem`.

The figure shows consumption policies computed by iteration of $K$ and $T$ respectively.

Chapter 3. Dynamic Programming
• 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
cp = ConsumerProblem()
v, c = initialize(cp)

@time bellman_operator(cp, v);
elapsed time: 0.095017748 seconds (24212168 bytes allocated, 30.48% gc time)

@time coleman_operator(cp, c);
elapsed time: 0.0696242 seconds (23937576 bytes allocated)
```

**Exercise 2**

Next lets 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 \texttt{linspace(0, 0.04, 4)}
- Consumption is plotted against assets for income shock fixed at the smallest value

The figure shows that higher interest rates boost savings and hence suppress consumption

**Exercise 3**

Now lets consider the long run asset levels held by households

Well take $r = 0.03$ and otherwise use default parameters

The following figure is a 45 degree diagram showing the law of motion for assets when consumption is optimal
m = ConsumerProblem(r=0.03, grid_max=4)
v_init, c_init = initialize(m)

c = compute_fixed_point(c -> coleman_operator(m, c),
                        c_init,
                        max_iter=150,
                        verbose=false)

a = m.asset_grid
R, z_vals = m.R, m.z_vals

# === generate savings plot === 
plot!(a, R * a + z_vals[1] - c[:, 1], label="Low income")
plot!(a, R * a + z_vals[2] - c[:, 2], label="High income")
plot!(xlabel="Current assets", ylabel="Next period assets")
plot!(xlabel="Current assets", ylabel="Next period assets")
The blue line and orange line represent the function

\[ a' = h(a, z) := Ra + z - c^*(a, z) \]

when income \( z \) takes its high and low values respectively.

The dashed line is the 45 degree line.

We can see from the figure that the dynamics will be stable assets do not diverge.

In fact there is a unique stationary distribution of assets that we can calculate by simulation:

- Can be proved via theorem 2 of [HP92]
- Represents the long run dispersion of assets across households when households have idiosyncratic shocks

Ergodicity is valid here, so stationary probabilities can be calculated by averaging over a single long time series.

- Hence to approximate the stationary distribution we can simulate a long time series for assets and histogram, as in the following figure.
Your task is to replicate the figure

- Parameters are as discussed above
- The histogram in the figure used a single time series \( \{a_t\} \) of length 500,000
- Given the length of this time series, the initial condition \((a_0, z_0)\) will not matter
- You might find it helpful to use the \texttt{MarkovChain} type from \texttt{quantecon}

**Exercise 4**

Following on from exercises 2 and 3, let's look at how savings and aggregate asset holdings vary with the interest rate

- Note: [LS18] section 18.6 can be consulted for more background on the topic treated in this exercise

For a given parameterization of the model, the mean of the stationary distribution can be interpreted as aggregate capital in an economy with a unit mass of \textit{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.

### 3.15.5 Solutions

#### Exercise 1

```julia
cp = ConsumerProblem()
K = 80
V, c = initialize(cp)
println("Starting value function iteration")
for i=1:K
    V = bellman_operator(cp, V)
end
c1 = bellman_operator(cp, V, ret_policy=true)

V2, c2 = initialize(cp)
println("Starting policy function iteration")
for i=1:K
    c2 = coleman_operator(cp, c2)
end

plot(cp.asset_grid, c1[:, 1], label="value function iteration")
plot!(cp.asset_grid, c2[:, 1], label="policy function iteration")
plot!(xlabel="asset level", ylabel="Consumption (low income)")
```

Starting value function iteration
Starting policy function iteration
Exercise 2

```julia
r_vals = linspace(0, 0.04, 4)
traces = []
legends = []

for r_val in r_vals
    cp = ConsumerProblem(r=r_val)
    v_init, c_init = initialize(cp)
    c = compute_fixed_point(x -> coleman_operator(cp, x),
                              c_init,
                              max_iter=150,
                              verbose=false)
    traces = push!(traces, c[:, 1])
    legends = push!(legends, "r = \$$(round(r_val, 3))\$"")
end

plot(traces, label=reshape(legends, 1, length(legends)))
plot!(xlabel="asset level", ylabel="Consumption (low income)")
```

3.15. Optimal Savings III: Occasionally Binding Constraints
Exercise 3

```julia
function compute_asset_series(cp, T=500000; verbose=false)
    # Simplify names
    z_idx = 1:length(z_vals)
    v_init, c_init = initialize(cp)
    c = compute_fixed_point(x -> coleman_operator(cp, x), c_init,
                             max_iter=150, verbose=false)

    cf = interp(cp.asset_grid, c)

    a = zeros(T+1)
    z_seq = simulate(MarkovChain(Π), T)
    for t=1:T
        i_z = z_seq[t]
        a[t+1] = R * a[t] + z_vals[i_z] - cf(a[t], i_z)
    end
    return a
end

cp = ConsumerProblem(r=0.03, grid_max=4)
```

550 Chapter 3. Dynamic Programming
\begin{verbatim}
a = compute_asset_series(cp) histogram(a, nbins=20, leg=false, normed=true, xlabel="assets")
\end{verbatim}

Exercise 4

\begin{verbatim}
M = 25
r_vals = linspace(0, 0.04, M)
xs = []
ys = []
legends = []
for b in [1.0, 3.0]
    asset_mean = zeros(M)
    for (i, r_val) in enumerate(r_vals)
        cp = ConsumerProblem(r=r_val, b=b)
        the_mean = mean(compute_asset_series(cp, 250000))
        asset_mean[i] = the_mean
    end
\end{verbatim}
xs = push!(xs, asset_mean)
y = push!(ys, r_vals)
legends = push!(legends, "b = \$b")
println("Finished iteration b=\$b")
end
plot(xs, ys, label=reshape(legends, 1, length(legends)))
plot!(xlabel="capital", ylabel="interest rate", yticks=[0, 0.045])

Finished iteration b=1.0
Finished iteration b=3.0

3.16 Robustness

Contents

- Robustness
3.16.1 Overview

This lecture modifies a Bellman equation to express a decision maker’s doubts about transition dynamics. His specification doubts make the decision maker want a robust decision rule. Robust means insensitive to misspecification of transition dynamics. The decision maker has a single approximating model. He calls it approximating to acknowledge that he doesn’t completely trust it. He fears that outcomes will actually be determined by another model that he cannot describe explicitly. All that he knows is that the actual data-generating model is in some (uncountable) set of models that surrounds his approximating model. He quantifies the discrepancy between his approximating model and the genuine data-generating model by using a quantity called entropy. (Well explain what entropy means below.) He wants a decision rule that will work well enough no matter which of those other models actually governs outcomes. This is what it means for his decision rule to be robust to misspecification of an approximating model. This may sound like too much to ask for, but . . . . . . a secret weapon is available to design robust decision rules. The secret weapon is max-min control theory. A value-maximizing decision maker enlists the aid of an (imaginary) value-minimizing model chooser to construct bounds on the value attained by a given decision rule under different models of the transition dynamics. The original decision maker uses those bounds to construct a decision rule with an assured performance level, no matter which model actually governs outcomes.
Note: In reading this lecture, please don’t think that our decision maker is paranoid when he conducts a worst-case analysis. By designing a rule that works well against a worst-case, his intention is to construct a rule that will work well across a set of models.

Sets of Models Imply Sets Of Values

Our robust decision maker wants to know how well a given rule will work when he does not know a single transition law . . .

. . . he wants to know sets of values that will be attained by a given decision rule $F$ under a set of transition laws

Ultimately, he wants to design a decision rule $F$ that shapes these sets of values in ways that he prefers

With this in mind, consider the following graph, which relates to a particular decision problem to be explained below

The figure shows a 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 makers approximating model
  
  – Entropy is zero when the set includes only the approximating model, indicating that the decision maker completely trusts the approximating model
  
  – Entropy is bigger, and the set of surrounding models is bigger, the less the decision maker trusts the approximating model

The shaded region indicates that for all models having entropy less than or equal to the number on the horizontal axis, the value obtained will be somewhere within the indicated set of values

Now let’s compare sets of values associated with two different decision rules, $F_r$ and $F_b$

In the next figure,

• The red set shows the value-entropy correspondence for decision rule $F_r$

• The blue set shows the value-entropy correspondence for decision rule $F_b$

The blue correspondence is skinnier than the red correspondence

This conveys the sense in which the decision rule $F_b$ is *more robust* than the decision rule $F_r$

• *More robust* means that the set of values is less sensitive to *increasing misspecification* as measured by entropy

3.16. *Robustness*
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 well explain in detail how to construct these sets of values for a given $F$, but for now . . .

Here is a hint about the secret weapons well use to construct these sets

- Well use some min problems to construct the lower bounds
- Well use some max problems to construct the upper bounds

We will also describe how to choose $F$ to shape the sets of values

This will involve crafting a skinnier set at the cost of a lower level (at least for low values of entropy)

**Inspiring Video**

If you want to understand more about why one serious quantitative researcher is interested in this approach, we recommend Lars Peter Hansens Nobel lecture

**Other References**

Our discussion in this lecture is based on

- [HS00]
- [HS08]

**3.16.2 The Model**

For simplicity, we present ideas in the context of a class of problems with linear transition laws and quadratic objective functions

To fit in with our earlier lecture on LQ control, we will treat loss minimization rather than value maximization

To begin, recall the infinite horizon LQ problem, where an agent chooses a sequence of controls $\{u_t\}$ to minimize

$$\sum_{t=0}^{\infty} \beta^t \{x_t'R_x + u_t'Q_u\}$$

subject to the linear law of motion

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \quad t = 0, 1, 2, \ldots$$

As before,

- $x_t$ is $n \times 1$, $A$ is $n \times n$
\[ u_t \text{ is } k \times 1, \quad B \text{ is } n \times k \]
\[ w_t \text{ is } j \times 1, \quad C \text{ is } n \times j \]
\[ R \text{ is } n \times n \quad \text{and} \quad Q \text{ is } k \times k \]

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 well quantify the quality of a model specification in terms of the maximal size of the expression

\[
\sum_{t=0}^{\infty} \beta^{t+1} w_{t+1}^t w_{t+1}
\]

### 3.16.3 Constructing More Robust Policies

If our agent takes \( \{w_t\} \) as a given deterministic sequence, then, drawing on intuition from earlier lectures on dynamic programming, we can anticipate Bellman equations such as

\[
J_{t-1}(x) = \min_{u} \{ x'Rx + u'Qu + \beta J_t(Ax + Bu + Cw_t) \}
\]

(Here \( J \) depends on \( t \) because the sequence \( \{w_t\} \) is not recursive)

Our tool for studying robustness is to construct a rule that works well even if an adverse sequence \( \{w_t\} \) occurs

In our framework, adverse means loss increasing

As well see, this will eventually lead us to construct the Bellman equation

\[
J(x) = \min_u \max_w \{ x'Rx + u'Qu + \beta [J(Ax + Bu + Cw) - \theta w'w] \}
\]  

(3.154)

Notice that weve added the penalty term \(-\theta w'w\)

Since \( w'w = \|w\|^2 \), this term becomes influential when \( w \) moves away from the origin

The penalty parameter \( \theta \) controls how much we penalize the maximizing agent for harming the minimizing agent

By raising \( \theta \) more and more, we more and more limit the ability of maximizing agent to distort outcomes relative to the approximating model

So bigger \( \theta \) is implicitly associated with smaller distortion sequences \( \{w_t\} \)

3.16. Robustness 557
Analyzing the Bellman equation

So what does $J$ in (3.154) look like?

As with the ordinary LQ control model, $J$ takes the form $J(x) = x' Px$ for some symmetric positive definite matrix $P$

One of our main tasks will be to analyze and compute the matrix $P$

Related tasks will be to study associated feedback rules for $u_t$ and $w_{t+1}$

First, using matrix calculus, you will be able to verify that

$$
\max_w \{(Ax + Bu + Cw)' P(Ax + Bu + Cw) - \theta w'w\}
= (Ax + Bu)' D(P)(Ax + Bu)
$$

where

$$
D(P) := P + PC(\theta I - C'PC)^{-1}C'P
$$

and $I$ is a $j \times j$ identity matrix. Substituting this expression for the maximum into (3.154) yields

$$
x'Px = \min_u \{x'R x + u'Qu + \beta (Ax + Bu)'D(P)(Ax + Bu)\}
$$

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 (3.157) and working through the algebra gives $x'Px = x'B(D(P))x$ for all $x$, or, equivalently,

$$
P = B(D(P))
$$

where $D$ is the operator defined in (3.156) and

$$
B(P) := R - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA + \beta A'PA
$$

The operator $B$ is the standard (i.e., non-robust) LQ Bellman operator, and $P = B(P)$ is the standard matrix Riccati equation coming from the Bellman equation see this discussion

Under some regularity conditions (see [HS08]), the operator $B \circ D$ has a unique positive definite fixed point, which we denote below by $\hat{P}$

A robust policy, indexed by $\theta$, is $u = -\hat{F}x$ where

$$
\hat{F} := (Q + \beta B'D(\hat{P})B)^{-1} \beta B'D(\hat{P})A
$$

We also define
\[ \hat{K} := (\theta I - C' \hat{P} C)^{-1} C' \hat{P} (A - B \hat{F}) \]  
(3.159)

The interpretation of \( \hat{K} \) is that \( w_{t+1} = \hat{K} x_t \) on the worst-case path of \( \{x_t\} \), in the sense that this vector is the maximizer of (3.155) evaluated at the fixed rule \( u = -\hat{F} x \)

Note that \( \hat{P}, \hat{F}, \hat{K} \) are all determined by the primitives and \( \theta \)

Note also that if \( \theta \) is very large, then \( D \) is approximately equal to the identity mapping

Hence, when \( \theta \) is large, \( \hat{P} \) and \( \hat{F} \) are approximately equal to their standard LQ values

Furthermore, when \( \theta \) is large, \( \hat{K} \) is approximately equal to zero

Conversely, smaller \( \theta \) is associated with greater fear of model misspecification, and greater concern for robustness

### 3.16.4 Robustness as Outcome of a Two-Person Zero-Sum Game

What we have done above can be interpreted in terms of a two-person zero-sum game in which \( \hat{F}, \hat{K} \) are Nash equilibrium objects

Agent 1 is our original agent, who seeks to minimize loss in the LQ program while admitting the possibility of misspecification

Agent 2 is an imaginary malevolent player

Agent 2’s malevolence helps the original agent to compute bounds on his value function across a set of models

We begin with agent 2’s problem

**Agent 2’s Problem**

Agent 2

1. knows a fixed policy \( \hat{F} \) specifying the behavior of agent 1, in the sense that \( u_t = -\hat{F} x_t \) for all \( t \)

2. responds by choosing a shock sequence \( \{w_t\} \) from a set of paths sufficiently close to the benchmark sequence \( \{0, 0, 0, \ldots\} \)

A natural way to say sufficiently close to the zero sequence is to restrict the summed inner product \( \sum_{t=1}^{\infty} w_t' w_t \) to be small

However, to obtain a time-invariant recursive formulation, it turns out to be convenient to restrict a discounted inner product

\[ \sum_{t=1}^{\infty} \beta^t w_t' w_t \leq \eta \]  
(3.160)
Now let \( F \) be a fixed policy, and let \( J_F(x_0, w) \) be the present-value cost of that policy given sequence \( w := \{w_t\} \) and initial condition \( x_0 \in \mathbb{R}^n \).

Substituting \(-Fx_t\) for \( u_t \) in (3.152), this value can be written as

\[
J_F(x_0, w) := \sum_{t=0}^{\infty} \beta^t x_t' (R + F'QF)x_t
\]

(3.161)

where

\[
x_{t+1} = (A - BF)x_t + Cw_{t+1}
\]

(3.162)

and the initial condition \( x_0 \) is as specified in the left side of (3.161)

Agent 2 chooses \( w \) to maximize agent 1s loss \( J_F(x_0, w) \) subject to (3.160)

Using a Lagrangian formulation, we can express this problem as

\[
\max_w \sum_{t=0}^{\infty} \beta^t \{ x_t' (R + F'QF)x_t - \beta \theta (w_{t+1} - \eta) \}
\]

where \( \{x_t\} \) satisfied (3.162) and \( \theta \) is a Lagrange multiplier on constraint (3.160)

For the moment, lets take \( \theta \) as fixed, allowing us to drop the constant \( \beta \theta \eta \) term in the objective function, and hence write the problem as

\[
\max_w \sum_{t=0}^{\infty} \beta^t \{ x_t' (R + F'QF)x_t - \beta w_{t+1}' w_{t+1} \}
\]

or, equivalently,

\[
\min_w \sum_{t=0}^{\infty} \beta^t \{ -x_t' (R + F'QF)x_t + \beta w_{t+1}' w_{t+1} \}
\]

(3.163)

subject to (3.162)

Whats striking about this optimization problem is that it is once again an LQ discounted dynamic programming problem, with \( w = \{w_t\} \) as the sequence of controls

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 2s problem is to set \( \theta \) to enforce the constraint (3.160), which can be done by choosing \( \theta = \theta_\eta \) such that

\[
\beta \sum_{t=0}^{\infty} \beta^t x_t' K(F, \theta_\eta)' K(F, \theta_\eta)x_t = \eta
\]

(3.164)

Here \( x_t \) is given by (3.162) which in this case becomes \( x_{t+1} = (A - BF + CK(F, \theta))x_t \)
Using Agent 2s Problem to Construct Bounds on the Value Sets

The Lower Bound

Define the minimized object on the right side of problem (3.163) as \( R_\theta(x_0, F) \).

Because minimizers minimize we have

\[
R_\theta(x_0, F) \leq \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t \right\} + \beta \sum_{t=0}^{\infty} \beta^t w_{t+1} \]

where \( x_{t+1} = (A - BF + CK(F, \theta))x_t \) and \( x_0 \) is a given initial condition.

This inequality in turn implies the inequality

\[
R_\theta(x_0, F) - \theta \text{ ent} \leq \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t \right\}
\]

where

\[
\text{ent} := \beta \sum_{t=0}^{\infty} \beta^t w_{t+1} \]

The left side of inequality (3.165) is a straight line with slope \(-\theta\)

Technically, it is a separating hyperplane

At a particular value of entropy, the line is tangent to the lower bound of values as a function of entropy.

In particular, the lower bound on the left side of (3.165) is attained when

\[
\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x_t'K(F, \theta)'K(F, \theta)x_t
\]

To construct the lower bound on the set of values associated with all perturbations \( w \) satisfying the entropy constraint (3.160) at a given entropy level, we proceed as follows:

- For a given \( \theta \), solve the minimization problem (3.163)
- Compute the minimizer \( R_\theta(x_0, F) \) and the associated entropy using (3.166)
- 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 (3.163) with the maximization problem

\[
V_{\tilde{\theta}}(x_0, F) = \max_w \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t - \beta \tilde{\theta} w_{t+1}'w_{t+1} \right\}
\] (3.167)

where now $\tilde{\theta} > 0$ penalizes the choice of $w$ with larger entropy.

(Notice that $\tilde{\theta} = -\theta$ in problem (3.163))

Because maximizers maximize we have

\[
V_{\tilde{\theta}}(x_0, F) \geq \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t \right\} - \beta \tilde{\theta} \sum_{t=0}^{\infty} \beta^t w_{t+1}'w_{t+1}
\]

which in turn implies the inequality

\[
V_{\tilde{\theta}}(x_0, F) + \tilde{\theta} \text{ent} \geq \sum_{t=0}^{\infty} \beta^t \left\{ -x_t'(R + F'QF)x_t \right\}
\] (3.168)

where

\[
\text{ent} \equiv \beta \sum_{t=0}^{\infty} \beta^t w_{t+1}'w_{t+1}
\]

The left side of inequality (3.168) is a straight line with slope $\tilde{\theta}$

The upper bound on the left side of (3.168) is attained when

\[
\text{ent} = \beta \sum_{t=0}^{\infty} \beta^t x_t'K(F, \tilde{\theta})'K(F, \tilde{\theta})x_t
\] (3.169)

To construct the upper bound on the set of values associated all perturbations $w$ with a given entropy we proceed much as we did for the lower bound

- For a given $\tilde{\theta}$, solve the maximization problem (3.167)
- Compute the maximizer $V_{\tilde{\theta}}(x_0, F)$ and the associated entropy using (3.169)
- Compute the upper bound on the value function $V_{\tilde{\theta}}(x_0, F) + \tilde{\theta} \text{ent}$ and plot it against $\text{ent}$
- 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

$$\min_{\{u_t\}} \sum_{t=0}^{\infty} \beta^t \left\{ x_t' Rx_t + u_t' Qu_t - \beta w_{t+1} w_{t+1} \right\}$$ (3.170)

where \( \{w_{t+1}\} \) satisfies \( w_{t+1} = K x_t \)

In other words, agent 1 minimizes

$$\sum_{t=0}^{\infty} \beta^t \left\{ x_t' (R - \beta \theta K' K) x_t + u_t' Qu_t \right\}$$ (3.171)

subject to

$$x_{t+1} = (A + CK) x_t + B u_t$$ (3.172)

Once again, the expression for the optimal policy can be found here we denote it by \( \tilde{F} \)

Nash Equilibrium

Clearly the \( \tilde{F} \) we have obtained depends on \( K \), which, in agent 2's problem, depended on an initial policy \( F \)

Holding all other parameters fixed, we can represent this relationship as a mapping \( \Phi \), where

$$\tilde{F} = \Phi(K(F, \theta))$$

The map \( F \mapsto \Phi(K(F, \theta)) \) corresponds to a situation in which

1. agent 1 uses an arbitrary initial policy \( F \)
2. agent 2 best responds to agent 1 by choosing \( K(F, \theta) \)
3. agent 1 best responds to agent 2 by choosing \( \tilde{F} = \Phi(K(F, \theta)) \)

As you may have already guessed, the robust policy \( \tilde{F} \) defined in (3.158) is a fixed point of the mapping \( \Phi \)

In particular, for any given \( \theta \),

1. \( K(\tilde{F}, \theta) = \tilde{K} \), where \( \tilde{K} \) is as given in (3.159)
2. \( \Phi(\tilde{K}) = \tilde{F} \)

A sketch of the proof is given in the appendix

3.16. Robustness
3.16.5 The Stochastic Case

Now we turn to the stochastic case, where the sequence \( \{w_t\} \) is treated as an iid sequence of random vectors. In this setting, we suppose that our agent is uncertain about the conditional probability distribution of \( w_{t+1} \). The agent takes the standard normal distribution \( N(0, I) \) as the baseline conditional distribution, while admitting the possibility that other nearby distributions prevail.

These alternative conditional distributions of \( w_{t+1} \) might depend nonlinearly on the history \( x_s, s \leq t \). To implement this idea, we need a notion of what it means for one distribution to be near another one. Here we adopt a very useful measure of closeness for distributions known as the relative entropy, or Kullback-Leibler divergence.

For densities \( p, q \), the Kullback-Leibler divergence of \( q \) from \( p \) is defined as:

\[
D_{KL}(p; q) := \int \ln \left( \frac{p(x)}{q(x)} \right) p(x) \, dx
\]

Using this notation, we replace (3.154) with the stochastic analogue

\[
J(x) = \min_u \max_{\psi \in P} \left\{ x'Rx + u'Qu + \beta \left[ \int (Ax + Bu + Cw) \psi(dw) - \theta D_{KL}(\psi, \phi) \right] \right\}
\]

Here \( P \) 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 (3.154), since it discourages large deviations from the benchmark.

Solving the Model

The maximization problem in (3.173) appears highly nontrivial, after all, we are maximizing over an infinite dimensional space consisting of the entire set of densities. However, it turns out that the solution is tractable, and in fact also falls within the class of normal distributions. First, we note that \( J \) has the form \( J(x) = x'Px + d \) for some positive definite matrix \( P \) and constant real number \( d \). Moreover, it turns out that if \( (I - \theta^{-1}C'PC)^{-1} \) is nonsingular, then

\[
\max_{\psi \in P} \left\{ \int (Ax + Bu + Cw)'P(Ax + Bu + Cw) \psi(dw) - \theta D_{KL}(\psi, \phi) \right\} = (Ax + Bu)'D(P)(Ax + Bu) + \kappa(\theta, P)
\]
where

\[ \kappa(\theta, P) := \theta \ln[\det(I - \theta^{-1}C'PC)^{-1}] \]

and the maximizer is the Gaussian distribution

\[ \psi = N \left( (\theta I - C'PC)^{-1}C'P(Ax + Bu), (I - \theta^{-1}C'PC)^{-1} \right) \]  

(3.175)

Substituting the expression for the maximum into Bellman equation (3.173) and using \( J(x) = x'Px + d \) gives

\[ x'Px + d = \min_u \left\{ x'Rx + u'Qu + \beta (Ax + Bu)'D(P)(Ax + Bu) + \beta [d + \kappa(\theta, P)] \right\} \]  

(3.176)

Since constant terms do not affect minimizers, the solution is the same as (3.157), leading to

\[ x'Px + d = x'B(D(P))x + \beta [d + \kappa(\theta, P)] \]

To solve this Bellman equation, we take \( \hat{P} \) to be the positive definite fixed point of \( B \circ D \)

In addition, we take \( \hat{d} \) as the real number solving \( d = \beta [d + \kappa(\theta, P)] \), which is

\[ \hat{d} := \frac{\beta}{1 - \beta}\kappa(\theta, P) \]  

(3.177)

The robust policy in this stochastic case is the minimizer in (3.176), which is once again \( u = -\hat{F}x \) for \( \hat{F} \) given by (3.158)

Substituting the robust policy into (3.175) we obtain the worst case shock distribution:

\[ w_{t+1} \sim N(\hat{K}x_t, (I - \theta^{-1}C'\hat{P}C)^{-1}) \]

where \( \hat{K} \) is given by (3.159)

Note that the mean of the worst-case shock distribution is equal to the same worst-case \( w_{t+1} \) as in the earlier deterministic setting

### Computing Other Quantities

Before turning to implementation, we briefly outline how to compute several other quantities of interest

### Worst-Case Value of a Policy

One thing we will be interested in doing is holding a policy fixed and computing the discounted loss associated with that policy
So let $F$ be a given policy and let $J_F(x)$ be the associated loss, which, by analogy with (3.173), satisfies

$$J_F(x) = \max_{\psi \in \mathcal{P}} \left\{ x'(R + F'QF)x + \beta \left[ \int J_F((A - BF)x + Cw) \psi(dw) - \theta D_{KL}(\psi, \phi) \right] \right\}$$

Writing $J_F(x) = x'P_Fx + d_F$ and applying the same argument used to derive (3.174) we get

$$x'P_Fx + d_F = x'(R + F'QF)x + \beta \left[ x'(A - BF)'D(P_F)(A - BF)x + d_F + \kappa(\theta, P_F) \right]$$

To solve this we take $P_F$ to be the fixed point

$$P_F = R + F'QF + \beta(A - BF)'D(P_F)(A - BF)$$

and

$$d_F := \frac{\beta}{1 - \beta} \kappa(\theta, P_F) = \frac{\beta}{1 - \beta} \theta \ln[\det(I - \theta^{-1}C'P_FC)^{-1}]$$

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 2s problem discussed above we use this in our computations.

### 3.16.6 Implementation

The QuantEcon.jl package provides a type called RBLQ for implementation of robust LQ optimal control.

The code can be found on GitHub.

Here is a brief description of the methods of the type

- `d_operator()` and `b_operator()` implement $D$ and $B$ respectively.
- `robust_rule()` and `robust_rule_simple()` both solve for the triple $\hat{F}, \hat{K}, \hat{P}$, as described in equations (3.158) – (3.159) and the surrounding discussion.
  - `robust_rule()` is more efficient.
  - `robust_rule_simple()` is more transparent and easier to follow.
- `K_to_F()` and `F_to_K()` solve the decision problems of agent 1 and agent 2 respectively.
- `compute_deterministic_entropy()` computes the left-hand side of (3.164).
- `evaluate_F()` computes the loss and entropy associated with a given policy see this discussion.

### 3.16.7 Application

Let us consider a monopolist similar to this one, but now facing model uncertainty.

The inverse demand function is $p_t = a_0 - a_1 y_t + d_t$.

where

$$d_{t+1} = \rho d_t + \sigma w_{t+1}, \quad \{w_t\} \overset{iid}{\sim} N(0, 1)$$
and all parameters are strictly positive

The period return function for the monopolist is

\[
    r_t = p_t y_t - \gamma \frac{(y_{t+1} - y_t)^2}{2} - cy_t
\]

Its objective is to maximize expected discounted profits, or, equivalently, to minimize \( E \sum_{t=0}^{\infty} \beta^t (-r_t) \)

To form a linear regulator problem, we take the state and control to be

\[
    x_t = \begin{bmatrix} 1 \\ y_t \\ d_t \end{bmatrix} \quad \text{and} \quad u_t = y_{t+1} - y_t
\]

Setting \( b := (a_0 - c)/2 \) we define

\[
    R = - \begin{bmatrix} 0 & b & 0 \\ b & -a_1 & 0 \\ 0 & 1/2 & 0 \end{bmatrix} \quad \text{and} \quad Q = \gamma/2
\]

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 agents baseline, with uncertainty parameterized by \( \theta \)

We compute value-entropy correspondences for two policies

1. The no concern for robustness policy \( F_0 \), which is the ordinary LQ loss minimizer
2. A moderate concern for robustness policy \( F_b \), with \( \theta = 0.02 \)

The code for producing the graph shown above, with blue being for the robust policy, is as follows

```julia
#=
@Author: Spencer Lyon <spencer.lyon@nyu.edu>
#=
using QuantEcon
using Plots
pyplot()

# model parameters
a_0 = 100
```
a_1 = 0.5
ρ = 0.9
σ_d = 0.05
β = 0.95
c = 2
γ = 50.0
θ = 0.002
ac = (a_0 - c) / 2.0

# Define LQ matrices
R = [ 0  ac  0;
    ac  -a_1  0.5;
    0.  0.5  0]
R = -R  # For minimization
Q = Matrix([γ / 2.0]')
A = [1.  0.  0.;
    0.1.  0.;
    0.  0.  ρ]
B = [0.1.  0.0;]
C = [0.0.  σ_d]'

## Functions

function evaluate_policy(θ::AbstractFloat, F::AbstractArray)
    rlq = RBLQ(Q, R, A, B, C, β, θ)
    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(TF<:AbstractFloat)(emax::AbstractFloat,
                                            F::AbstractArray{TF},
                                            bw::String,
                                            grid_size::Integer=1000)
    if lowercase(bw) == "worst"
        θs = 1. / linspace(1e-8, 1000, grid_size)
    else
        θs = -1. / linspace(1e-8, 1000, grid_size)
    end

data = Array{TF}(grid_size, 2)

    for (i, θ) in enumerate(θs)
        data[i, :) = collect(evaluate_policy(θ, F))
        if data[i, 2] >= emax  # stop at this entropy level
            data = data[1:i, :]
            break
        end
    end
return data
## Main

# compute optimal rule
optimal_lq = LQ(Q, R, A, B, C, zero(B' * A), bet=β)
Po, Fo, Do = stationary_values(optimal_lq)

# compute robust rule for our θ
baseline_robust = RBLQ(Q, R, A, B, C, β, θ)
Fb, Kb, Pb = robust_rule(baseline_robust)

# Check the positive definiteness of worst-case covariance matrix to
# ensure that θ exceeds the breakdown point

test_matrix = eye(size(Pb, 1)) - (C' * Pb + C ./ θ)[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")

# 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 data_pair in data_pairs
    for data in data_pair
        x, y = data[:, 2], data[:, 1]
        curve = LinInterp(x, y)
        push!(egrid_data, curve.(egrid))
    end
end

plot(egrid, egrid_data, color=[:red :red :blue :blue])
plot!(egrid, egrid_data[1], fillrange=egrid_data[2],
    fillcolor=:red, fillalpha=0.1, color=:red, legend=:none)
plot!(egrid, egrid_data[3], fillrange=egrid_data[4],
    fillcolor=:blue, fillalpha=0.1, color=:blue, legend=:none)
plot!(xlabel="Entropy", ylabel="Value")

Here is another such figure, with θ = 0.002 instead of 0.02
Can you explain the different shape of the value-entropy correspondence for the robust policy?

### 3.16.8 Appendix

We sketch the proof only of the first claim in this section, which is that, for any given $\theta$, $K(\hat{F}, \theta) = \hat{K}$, where $\hat{K}$ is as given in (3.159)

This is the content of the next lemma

**Lemma.** If $\hat{P}$ is the fixed point of the map $B \circ D$ and $\hat{F}$ is the robust policy as given in (3.158), then

$$K(\hat{F}, \theta) = (\theta I - C' \hat{P} C)^{-1} C' \hat{P} (A - B \hat{F})$$  \hfill (3.179)

**Proof:** As a first step, observe that when $F = \hat{F}$, the Bellman equation associated with the LQ problem (3.162) – (3.163) is

$$\hat{P} = -R - \hat{F}' Q \hat{F} - \beta^2 (A - B \hat{F})' \hat{P} C (\beta \theta I + \beta C' \hat{P} C)^{-1} C' \hat{P} (A - B \hat{F}) + \beta (A - B \hat{F})' \hat{P} (A - B \hat{F})$$  \hfill (3.180)

(revisit this discussion if you dont know where (3.180) comes from) and the optimal policy is

$$w_{t+1} = -\beta (\beta \theta I + \beta C' \hat{P} C)^{-1} C' \hat{P} (A - B \hat{F}) x_t$$

Suppose for a moment that $-\hat{P}$ solves the Bellman equation (3.180)
In this case the policy becomes

$$w_{t+1} = (\theta I - C' \hat{P} C)^{-1} C' \hat{P} (A - B \hat{F}) x_t$$

which is exactly the claim in (3.179)

Hence it remains only to show that $-\hat{P}$ solves (3.180), or, in other words,

$$\hat{P} = R + \hat{F}' Q \hat{F} + \beta (A - B \hat{F})' \hat{P} C (\theta I - C' \hat{P} C)^{-1} C' \hat{P} (A - B \hat{F}) + \beta (A - B \hat{F})' \hat{P} (A - B \hat{F})$$

Using the definition of $D$, we can rewrite the right-hand side more simply as

$$R + \hat{F}' Q \hat{F} + \beta (A - B \hat{F})' D (\hat{P}) (A - B \hat{F})$$

Although it involves a substantial amount of algebra, it can be shown that the latter is just $\hat{P}$

(Hint: Use the fact that $\hat{P} = B(D(\hat{P}))$)

### 3.17 Discrete State Dynamic Programming

#### 3.17.1 Overview

In this lecture we discuss a family of dynamic programming problems with the following features:

1. a discrete state space and discrete choices (actions)
2. an infinite horizon
3. discounted rewards
4. Markov state transitions

We call such problems discrete dynamic programs, or discrete DPs

Discrete DPs are the workhorses in much of modern quantitative economics, including
• monetary economics
• search and labor economics
• household savings and consumption theory
• investment theory
• asset pricing
• industrial organization, etc.

When a given model is not inherently discrete, it is common to replace it with a discretized version in order to use discrete DP techniques.

This lecture covers
• the theory of dynamic programming in a discrete setting, plus examples and applications
• a powerful set of routines for solving discrete DPs from the QuantEcon code library

**How to Read this Lecture**

We use dynamic programming many applied lectures, such as
• The shortest path lecture
• The McCall search model lecture
• The optimal growth lecture

The objective of this lecture is to provide a more systematic and theoretical treatment, including algorithms and implementation, while focusing on the discrete case.

The code discussed below was authored primarily by Daisuke Oyama.

**References**

For background reading on dynamic programming and additional applications, see, for example,

• [LS18]
• [HLL96], section 3.5
• [Put05]
• [SLP89]
• [Rus96]
• [MF02]
• EDTC, chapter 5
3.17.2 Discrete DPs

Loosely speaking, a discrete DP is a maximization problem with an objective function of the form

\[
\mathbb{E} \sum_{t=0}^{\infty} \beta^t r(s_t, a_t)
\]

(3.181)

where

- \(s_t\) is the state variable
- \(a_t\) is the action
- \(\beta\) is a discount factor
- \(r(s_t, a_t)\) is interpreted as a current reward when the state is \(s_t\) and the action chosen is \(a_t\)

Each pair \((s_t, a_t)\) pins down transition probabilities \(Q(s_t, a_t, s_{t+1})\) for the next period state \(s_{t+1}\)

Thus, actions influence not only current rewards but also the future time path of the state

The essence of dynamic programming problems is to trade off current rewards vs favorable positioning of the future state (modulo randomness)

Examples:

- consuming today vs saving and accumulating assets
- accepting a job offer today vs seeking a better one in the future
- exercising an option now vs waiting

Policies

The most fruitful way to think about solutions to discrete DP problems is to compare policies

In general, a policy is a randomized map from past actions and states to current action

In the setting formalized below, it suffices to consider so-called stationary Markov policies, which consider only the current state

In particular, a stationary Markov policy is a map \(\sigma\) from states to actions

\[ a_t = \sigma(s_t) \]

indicates that \(a_t\) is the action to be taken in state \(s_t\)

It is known that, for any arbitrary policy, there exists a stationary Markov policy that dominates it at least weakly

- See section 5.5 of [Put05] for discussion and proofs

In what follows, stationary Markov policies are referred to simply as policies

The aim is to find an optimal policy, in the sense of one that maximizes (3.181)

Let’s now step through these ideas more carefully
Formal definition

Formally, a discrete dynamic program consists of the following components:

1. A finite set of states $S = \{0, \ldots, n-1\}$
2. A finite set of feasible actions $A(s)$ for each state $s \in S$, and a corresponding set of feasible state-action pairs

$$SA := \{(s, a) \mid s \in S, a \in A(s)\}$$

3. A reward function $r: SA \to \mathbb{R}$
4. A transition probability function $Q: SA \to \Delta(S)$, where $\Delta(S)$ is the set of probability distributions over $S$
5. A discount factor $\beta \in [0, 1)$

We also use the notation $A := \bigcup_{s \in S} A(s) = \{0, \ldots, m-1\}$ and call this set the action space.

A policy is a function $\sigma: S \to A$.

A policy is called feasible if it satisfies $\sigma(s) \in A(s)$ for all $s \in S$.

Denote the set of all feasible policies by $\Sigma$.

If a decision maker uses a policy $\sigma \in \Sigma$, then
- the current reward at time $t$ is $r(s_t, \sigma(s_t))$
- the probability that $s_{t+1} = s'$ is $Q(s_t, \sigma(s_t), s')$

For each $\sigma \in \Sigma$, define
- $r_\sigma$ by $r_\sigma(s) := r(s, \sigma(s))$
- $Q_\sigma$ by $Q_\sigma(s, s') := Q(s, \sigma(s), s')$

Notice that $Q_\sigma$ is a stochastic matrix on $S$.

It gives transition probabilities of the controlled chain when we follow policy $\sigma$.

If we think of $r_\sigma$ as a column vector, then so is $Q_\sigma^t r_\sigma$, and the $s$-th row of the latter has the interpretation

$$\left(Q_\sigma^t r_\sigma\right)(s) = \mathbb{E}[r(s_t, \sigma(s_t)) \mid s_0 = s] \quad \text{when } \{s_t\} \sim Q_\sigma$$ (3.182)

Comments
- $\{s_t\} \sim Q_\sigma$ means that the state is generated by stochastic matrix $Q_\sigma$
- See this discussion on computing expectations of Markov chains for an explanation of the expression in (3.182)

Notice that we are not really distinguishing between functions from $S$ to $\mathbb{R}$ and vectors in $\mathbb{R}^n$.

This is natural because they are in one to one correspondence.
Value and Optimality

Let $v_\sigma(s)$ denote the discounted sum of expected reward flows from policy $\sigma$ when the initial state is $s$.

To calculate this quantity we pass the expectation through the sum in (3.181) and use (3.182) to get

$$v_\sigma(s) = \sum_{t=0}^{\infty} \beta^t (Q^t r_\sigma)(s) \quad (s \in S)$$

This function is called the policy value function for the policy $\sigma$.

The optimal value function, or simply value function, is the function $v^* : S \to \mathbb{R}$ defined by

$$v^*(s) = \max_{\sigma \in \Sigma} v_\sigma(s) \quad (s \in S)$$

(We can use max rather than sup here because the domain is a finite set.)

A policy $\sigma \in \Sigma$ is called optimal if $v_\sigma(s) = v^*(s)$ for all $s \in S$.

Given any $w : S \to \mathbb{R}$, a policy $\sigma \in \Sigma$ is called $w$-greedy if

$$\sigma(s) \in \arg\max_{a \in A(s)} \left\{ r(s,a) + \beta \sum_{s' \in S} w(s') Q(s,a,s') \right\} \quad (s \in S)$$

As discussed in detail below, optimal policies are precisely those that are $v^*$-greedy.

Two Operators

It is useful to define the following operators:

- The Bellman operator $T : \mathbb{R}^S \to \mathbb{R}^S$ is defined by
  $$ (Tv)(s) = \max_{a \in A(s)} \left\{ r(s,a) + \beta \sum_{s' \in S} v(s') Q(s,a,s') \right\} \quad (s \in S) $$

- For any policy function $\sigma \in \Sigma$, the operator $T_\sigma : \mathbb{R}^S \to \mathbb{R}^S$ is defined by
  $$ (T_\sigma v)(s) = r(s,\sigma(s)) + \beta \sum_{s' \in S} v(s') Q(s,\sigma(s),s') \quad (s \in S) $$

This can be written more succinctly in operator notation as

$$T_\sigma v = r_\sigma + \beta Q_\sigma v$$

The two operators are both monotone

- $v \leq w$ implies $Tv \leq Tw$ pointwise on $S$, and similarly for $T_\sigma$

They are also contraction mappings with modulus $\beta$

- $\|Tv - Tw\| \leq \beta \|v - w\|$ and similarly for $T_\sigma$, where $\|\cdot\|$ is the max norm

For any policy $\sigma$, its value $v_\sigma$ is the unique fixed point of $T_\sigma$

For proofs of these results and those in the next section, see, for example, EDTC, chapter 10
The Bellman Equation and the Principle of Optimality

The main principle of the theory of dynamic programming is that

- the optimal value function $v^*$ is a unique solution to the Bellman equation,

$$
v(s) = \max_{a \in A(s)} \left\{ r(s, a) + \beta \sum_{s' \in S} v(s') Q(s, a, s') \right\} \quad (s \in S),
$$

or in other words, $v^*$ is the unique fixed point of $T$, and

- $\sigma^*$ is an optimal policy function if and only if it is $v^*$-greedy

By the definition of greedy policies given above, this means that

$$
\sigma^*(s) \in \arg \max_{a \in A(s)} \left\{ r(s, a) + \beta \sum_{s' \in S} v^*(s') Q(s, \sigma(s), s') \right\} \quad (s \in S)
$$

3.17.3 Solving Discrete DPs

Now that the theory has been set out, let’s turn to solution methods

Code for solving discrete DPs is available in ddp.jl from the QuantEcon.jl code library

It implements the three most important solution methods for discrete dynamic programs, namely

- value function iteration
- policy function iteration
- modified policy function iteration

Let’s briefly review these algorithms and their implementation

Value Function Iteration

Perhaps the most familiar method for solving all manner of dynamic programs is value function iteration

This algorithm uses the fact that the Bellman operator $T$ is a contraction mapping with fixed point $v^*$

Hence, iterative application of $T$ to any initial function $v^0 : S \to \mathbb{R}$ converges to $v^*$

The details of the algorithm can be found in the appendix

Policy Function Iteration

This routine, also known as Howards policy improvement algorithm, exploits more closely the particular structure of a discrete DP problem

Each iteration consists of
1. A policy evaluation step that computes the value $v_\sigma$ of a policy $\sigma$ by solving the linear equation $v = T_\sigma v$

2. A policy improvement step that computes a $v_\sigma$-greedy policy

In the current setting policy iteration computes an exact optimal policy in finitely many iterations

- See theorem 10.2.6 of EDTC for a proof

The details of the algorithm can be found in the appendix

**Modified Policy Function Iteration**

Modified policy iteration replaces the policy evaluation step in policy iteration with partial policy evaluation

The latter computes an approximation to the value of a policy $\sigma$ by iterating $T_\sigma$ for a specified number of times

This approach can be useful when the state space is very large and the linear system in the policy evaluation step of policy iteration is correspondingly difficult to solve

The details of the algorithm can be found in the appendix

### 3.17.4 Example: A Growth Model

Let’s consider a simple consumption-saving model

A single household either consumes or stores its own output of a single consumption good

The household starts each period with current stock $s$

Next, the household chooses a quantity $a$ to store and consumes $c = s - a$

- Storage is limited by a global upper bound $M$
- Flow utility is $u(c) = e^c$

Output is drawn from a discrete uniform distribution on $\{0, \ldots, B\}$

The next period stock is therefore

$$s' = a + U$$

where $U \sim U[0, \ldots, B]$

The discount factor is $\beta \in [0, 1)$

**Discrete DP Representation**

We want to represent this model in the format of a discrete dynamic program

To this end, we take

- the state variable to be the stock $s$
- the state space to be $S = \{0, \ldots, M + B\}$
\[ n = M + B + 1 \]

- hence \( n = M + B + 1 \)

- the action to be the storage quantity \( a \)
- the set of feasible actions at \( s \) to be \( A(s) = \{0, \ldots, \min\{s, M\}\} \)
  - hence \( A = \{0, \ldots, M\} \) and \( m = M + 1 \)
- the reward function to be \( r(s, a) = u(s - a) \)
- the transition probabilities to be
  \[
  Q(s, a, s') := \begin{cases} 
  \frac{1}{B+1} & \text{if } a \leq s' \leq a + B \\
  0 & \text{otherwise}
  \end{cases}
  \]  

(3.183)

**Defining a DiscreteDP Instance**

This information will be used to create an instance of DiscreteDP by passing the following information

1. An \( n \times m \) reward array \( R \)
2. An \( n \times m \times n \) transition probability array \( Q \)
3. A discount factor \( \beta \)

For \( R \) we set \( R[s, a] = u(s - a) \) if \( a \leq s \) and \(-\infty\) otherwise

For \( Q \) we follow the rule in (3.183)

Note:

- The feasibility constraint is embedded into \( R \) by setting \( R[s, a] = -\infty \) for \( a \notin A(s) \)
- Probability distributions for \((s, a)\) with \( a \notin A(s) \) can be arbitrary

The following code sets up these objects for us

```julia
struct SimpleOG{TI <: Integer, T <: Real, TR <: AbstractArray{T}, TQ <: AbstractArray{T}}
    B :: TI
    M :: TI
    \(\alpha\) :: T
    \(\beta\) :: T
    R :: TR
    Q :: TQ
end

function SimpleOG{T <: Real}(;B::Integer=10, M::Integer=5, \(\alpha\)::T=0.5, \(\beta\)::T=0.9)
    \(u(c) = c^{\alpha}\)
    \(n = B + M + 1\)
    \(m = M + 1\)

    R = Matrix{T}(n, m)
    Q = zeros(Float64, n, m, n)
```

578 Chapter 3. Dynamic Programming
for a in 0:M
    Q[:, a + 1, (a:(a + B)) + 1] = 1 / (B + 1)
for s in 0:(B + M)
    R[s + 1, a + 1] = a<=s ? u(s - a) : -Inf
end
end
return SimpleOG(B, M, a, β, R, Q)
end

Lets run this code and create an instance of SimpleOG

```python

```
g = SimpleOG()

```

Instances of DiscreteDP are created using the signature DiscreteDP(R, Q, β)

Lets create an instance using the objects stored in g

```python

```
using QuantEcon
ddp = DiscreteDP(g.R, g.Q, g.β)

```

Now that we have an instance ddp of DiscreteDP we can solve it as follows

```python

```
results = solve(ddp, PFI)

```

Lets see what we've got here

```python

```
fieldnames(results)

```

5-element Array(Symbol,1):
 :v
 :Tv
 :num_iter
 :sigma
 :mc

The most important attributes are v, the value function, and σ, the optimal policy

```python

```
results.v

```

16-element Array(Float64,1):
 19.0174
 20.0174
 20.4316
 20.7495
 21.0408
 21.3087
 21.5448
 21.7693
 21.9827
 22.1882

3.17. Discrete State Dynamic Programming
Here 1 is subtracted from `results.sigma` because we added 1 to each state and action to create valid indices

Since we've used policy iteration, these results will be exact unless we hit the iteration bound `max_iter`.

Let's make sure this didn't happen

```julia
results.num_iter
```

3

In this case we converged in only 3 iterations.

Another interesting object is `results.mc`, which is the controlled chain defined by $Q_{\sigma^*}$, where $\sigma^*$ is the optimal policy.

In other words, it gives the dynamics of the state when the agent follows the optimal policy.

Since this object is an instance of `MarkovChain` from `QuantEcon.jl` (see this lecture for more discussion), we can easily simulate it, compute its stationary distribution and so on.

```julia
stationary_distributions(results.mc)[1]
```

16-element Array{Float64,1}:
0.0173219
0.0412106
0.0412106
0.0412106
0.0412106
0.0412106
0.0412106
0.0412106
0.0412106
0.0412106
0.0412106
0.0412106
0.0412106
0.0412106
0.0412106
0.0412106
0.0412106
What happens if the agent is more patient?

\[
g_2 = \text{SimpleOG} (\beta=0.99)
\]

\[
ddp_2 = \text{DiscreteDP}(g_2.R, g_2.Q, g_2.\beta)
\]

\[
\text{results}_2 = \text{solve}(ddp_2, \text{PFI})
\]

\[
\text{std}_2 = \text{stationary_distributions}(\text{results}_2.\text{mc})[1]
\]

16-element Array{Float64,1}:
0.00546913
0.0232134
0.0314779
0.0480068
0.0562713
0.0909091
0.0909091
0.0909091
0.0909091
0.0909091
0.0909091
0.08544
0.0676957
0.0594312
0.0429023
0.0346378

```julia
using Plots
pyplot()
bar(std_2, label="stationary dist")
```

If we look at the bar graph we can see the rightward shift in probability mass

**State-Action Pair Formulation**

The `DiscreteDP` type in fact provides a second interface to setting up an instance

One of the advantages of this alternative set up is that it permits use of a sparse matrix for $Q$

(An example of using sparse matrices is given in the exercises below)
The call signature of the second formulation is `DiscreteDP(R, Q, β, s_indices, a_indices)` where

- `s_indices` and `a_indices` are arrays of equal length L enumerating all feasible state-action pairs
- `R` is an array of length L giving corresponding rewards
- `Q` is an L x n transition probability array

Here's how we could set up these objects for the preceding example

```julia
using QuantEcon

B = 10
M = 5
α = 0.5
β = 0.9
u(c) = c^α
n = B + M + 1
m = M + 1

s_indices = Int64[]
a_indices = Int64[]
Q = Array{Float64}(0, n)
R = Float64[]

b = 1.0 / (B + 1)

for s in 0:(M + B)
    for a in 0:min(M, s)
        s_indices = [s_indices; s + 1]
        a_indices = [a_indices; a + 1]
        q = zeros(Float64, 1, n)
        q[(a + 1):((a + B) + 1)] = b
        Q = [Q; q]
        R = [R; u(s-a)]
    end
end

ddp = DiscreteDP(R, Q, β, s_indices, a_indices);
results = solve(ddp, PFI)
```

### 3.17.5 Exercises

In the stochastic optimal growth lecture dynamic programming lecture, we solve a benchmark model that has an analytical solution to check we could replicate it numerically.

The exercise is to replicate this solution using `DiscreteDP`.

### 3.17.6 Solutions

Written jointly with Diasuke Oyama and Max Huber
Setup

Details of the model can be found in the lecture. As in the lecture, we let \( f(k) = k^\alpha \) with \( \alpha = 0.65 \), \( u(c) = \log c \), and \( \beta = 0.95 \).

\[
\begin{align*}
\alpha &= 0.65 \\
f(k) &= k^\alpha \\
u_{\log}(x) &= \log(x) \\
\beta &= 0.95
\end{align*}
\]

Here we want to solve a finite state version of the continuous state model above. We discretize the state space into a grid of size \( \text{grid\_size}=500 \), from \( 10^{-6} \) to \( \text{grid\_max}=2 \).

\[
\begin{align*}
\text{grid\_max} &= 2 \\
\text{grid\_size} &= 500 \\
\text{grid} &= \text{linspace}(1e-6, \text{grid\_max}, \text{grid\_size})
\end{align*}
\]

We choose the action to be the amount of capital to save for the next period (the state is the capital stock at the beginning of the period). Thus the state indices and the action indices are both \( 1, \text{grid\_size} \). Action (indexed by) \( a \) is feasible at state (indexed by) \( s \) if and only if \( \text{grid}[a] < f([\text{grid}[s]]) \) (zero consumption is not allowed because of the log utility).

Thus the Bellman equation is:

\[
v(k) = \max_{0 < k' < f(k)} u(f(k) - k') + \beta v(k'),
\]

where \( k' \) is the capital stock in the next period.

The transition probability array \( Q \) will be highly sparse (in fact it is degenerate as the model is deterministic), so we formulate the problem with state-action pairs, to represent \( Q \) in sparse matrix format.

We first construct indices for state-action pairs:

\[
\begin{align*}
C &= f.(\text{grid}) .- \text{grid'} \\
\text{coord} &= \text{repmat} (\text{collect} (1:\text{grid\_size}), 1, \text{grid\_size}) \quad \# \text{coordinate matrix} \\
\text{s\_indices} &= \text{coord}[C > 0] \\
\text{a\_indices} &= \text{transpose} (\text{coord}) [C > 0] \\
L &= \text{length} (\text{a\_indices})
\end{align*}
\]

118841

\[
\begin{align*}
\text{s\_indices}
\end{align*}
\]

118841-element Array{Int64,1}:

1
2
Now lets set up $R$ and $Q$

```julia
R = u_log.(C[C.>0])

Q = spzeros(L, grid_size)
for i in 1:L
    Q[i, a_indices[i]] = 1
end
```

Were now in a position to create an instance of `DiscreteDP` corresponding to the growth model.

```julia
ddp = DiscreteDP(R, Q, β, s_indices, a_indices)
```

**Solving the Model**

```julia
results = solve(ddp, PFI)

v, σ, num_iter = results.v, results.sigma, results.num_iter
num_iter
```

10

Let us compare the solution of the discrete model with the exact solution of the original continuous model. Here’s the exact solution:
\[ c = f(\text{grid}) - \text{grid}[\sigma] \]
\[ ab = \alpha \times \beta \]
\[ c1 = (\log(1 - \alpha \times \beta) + \log(\alpha \times \beta) \times \alpha + \beta / (1 - \alpha \times \beta)) / (1 - \beta) \]
\[ c2 = \alpha / (1 - \alpha \times \beta) \]

\[ v_\text{star}(k) = c1 + c2 \times \log(k) \]
\[ c_\text{star}(k) = (1 - \alpha + \beta) \times k^{\alpha} \]

\text{c\_star (generic function with 1 method)}

Let's plot the value functions.

\text{plot(grid, [v v\_star.(grid)], ylim=(-40, -32), lw=2, label=["discrete" \rightarrow "continuous"])}

They are barely distinguishable (although you can see the difference if you zoom).

Now let's look at the discrete and exact policy functions for consumption.
These functions are again close, although some difference is visible and becomes more obvious as you zoom. Here are some statistics:

```plaintext
maximum(abs, v - v_star.(grid))
```

121.49819147053377

This is a big error, but most of the error occurs at the lowest gridpoint. Otherwise the fit is reasonable:

```plaintext
maximum(abs, (v - v_star.(grid))[2:end])
```

0.012681735127422655

The value function is monotone, as expected:

```plaintext
all(diff(v).>=0)
```

true
Comparison of the solution methods

Let's try different solution methods. The results below show that policy function iteration and modified policy function iteration are much faster than value function iteration.

```julia
@time results = solve(ddp, PFI)
0.781454 seconds (573 allocations: 108.376 MB, 2.93% gc time)

@time res1 = solve(ddp, VFI, max_iter=500, epsilon=1e-4)
10.217790 seconds (116.72 k allocations: 810.294 MB, 1.43% gc time)

res1.num_iter
294

σ == res1.sigma
true

@time res2 = solve(ddp, MPFI, max_iter=500, epsilon=1e-4)
1.523519 seconds (273.77 k allocations: 664.077 MB, 5.73% gc time)

res2.num_iter
16

σ == res2.sigma
true
```

Replication of the figures

Let's visualize convergence of value function iteration, as in the lecture.

```julia
w_init = 5 * log.(grid) - 25  # Initial condition
n = 50
ws = []
colors = []
w = w_init
for i in 0:n-1
    w = bellman_operator(ddp, w)
    push!(ws, w)
```
We next plot the consumption policies along the value iteration. First we write a function to generate the and record the policies at given stages of iteration.

```julia
function compute_policies(n_vals)  
c_policies = []  
w = w_init  
for n in 1:maximum(n_vals)  
w = bellman_operator(ddp, w)  
if n in n_vals  
  σ = compute_greedy(ddp, w)  
c_policy = f(grid) - grid[σ]  
push!(c_policies, c_policy)  
end  
end
```

3.17. Discrete State Dynamic Programming
Now let's generate the plots.

```julia
true_c = c_star.(grid)
c_policies = compute_policies(2, 4, 6)
l1 = "approximate optimal policy"
l2 = "optimal consumption policy"
labels = [l1 l1 l1 l2 l2 l2]
plot(grid,
     plot_vecs,
     xlim=(0, 2),
     ylim=(0, 1),
     layout=(3, 1),
     lw=2,
     label=labels,
     size=(600, 800),
     title=\"2 iterations\" \"4 iterations\" \"6 iterations\")
```
3.17. Discrete State Dynamic Programming
Dynamics of the capital stock

Finally, let us work on Exercise 2, where we plot the trajectories of the capital stock for three different discount factors, 0.9, 0.94, and 0.98, with initial condition $k_0 = 0.1$.

```julia
discount_factors = (0.9, 0.94, 0.98)
k_init = 0.1

k_init_ind = findfirst(collect(grid) .>= k_init, true)

sample_size = 25

ddp0 = DiscreteDP(R, Q, β, s_indices, a_indices)
k_paths = []
labels = []

for β in discount_factors
    ddp0.beta = β
    res0 = solve(ddp0, PFI)
    k_path_ind = simulate(res0.mc, sample_size, init=k_init_ind)
    k_path = grid[k_path_ind.+1]
    push!(k_paths, k_path)
    push!(labels, "β = $β")
end

plot(k_paths,
     xlabel="time",
     ylabel="capital",
     ylim=(0.1, 0.3),
     lw=2,
     markershape=:circle,
     label=reshape(labels, 1, length(labels)))
```

592 Chapter 3. Dynamic Programming
This appendix covers the details of the solution algorithms implemented for DiscreteDP. We will make use of the following notions of approximate optimality:

- For $\varepsilon > 0$, $v$ is called an $\varepsilon$-approximation of $v^*$ if $\|v - v^*\| < \varepsilon$
- A policy $\sigma \in \Sigma$ is called $\varepsilon$-optimal if $v_{\sigma}$ is an $\varepsilon$-approximation of $v^*$

**Value Iteration**

The DiscreteDP value iteration method implements value function iteration as follows

1. Choose any $v^0 \in \mathbb{R}^n$, and specify $\varepsilon > 0$; set $i = 0$
2. Compute $v^{i+1} = T v^i$
3. If $\|v^{i+1} - v^i\| < [(1 - \beta)/(2\beta)]\varepsilon$, then go to step 4; otherwise, set $i = i + 1$ and go to step 2
4. Compute a $v_i^{i+1}$-greedy policy $\sigma$, and return $v_i^{i+1}$ and $\sigma$

Given $\varepsilon > 0$, the value iteration algorithm

- terminates in a finite number of iterations
- returns an $\varepsilon/2$-approximation of the optimal value function and an $\varepsilon$-optimal policy function (unless iter_max is reached)
(While not explicit, in the actual implementation each algorithm is terminated if the number of iterations reaches \texttt{iter\_max})

**Policy Iteration**

The \texttt{DiscreteDP} policy iteration method runs as follows:

1. Choose any \( v^0 \in \mathbb{R}^n \) and compute a \( v^0 \)-greedy policy \( \sigma^0 \); set \( i = 0 \)
2. Compute the value \( v_{\sigma^i} \) by solving the equation \( v = T_{\sigma^i} v \)
3. Compute a \( v_{\sigma^i} \)-greedy policy \( \sigma^{i+1} \); let \( \sigma^{i+1} = \sigma^i \) if possible
4. If \( \sigma^{i+1} = \sigma^i \), then return \( v_{\sigma^i} \) and \( \sigma^{i+1} \); otherwise, set \( i = i + 1 \) and go to step 2

The policy iteration algorithm terminates in a finite number of iterations

It returns an optimal value function and an optimal policy function (unless \texttt{iter\_max} is reached)

**Modified Policy Iteration**

The \texttt{DiscreteDP} modified policy iteration method runs as follows:

1. Choose any \( v^0 \in \mathbb{R}^n \), and specify \( \varepsilon > 0 \) and \( k \geq 0 \); set \( i = 0 \)
2. Compute a \( v^i \)-greedy policy \( \sigma^{i+1} \); let \( \sigma^{i+1} = \sigma^i \) if possible (for \( i \geq 1 \))
3. Compute \( u = T v^i \) \((= T_{\sigma^{i+1}} v^i)\). If \( \text{span}(u - v^i) < [(1 - \beta)/\beta]\varepsilon \), then go to step 5; otherwise go to step 4
   - Span is defined by \( \text{span}(z) = \max(z) - \min(z) \)
4. Compute \( v^{i+1} = (T_{\sigma^{i+1}})^k u \) \((= (T_{\sigma^{i+1}})^{k+1} v^i)\); set \( i = i + 1 \) and go to step 2
5. Return \( v = u + [\beta/(1 - \beta)][(\min(u - v^i) + \max(u - v^i))/2]\mathbf{1} \) and \( \sigma_{i+1} \)

Given \( \varepsilon > 0 \), provided that \( v^0 \) is such that \( T v^0 \geq v^0 \), the modified policy iteration algorithm terminates in a finite number of iterations

It returns an \( \varepsilon/2 \)-approximation of the optimal value function and an \( \varepsilon \)-optimal policy function (unless \texttt{iter\_max} is reached).

See also the documentation for \texttt{DiscreteDP}
These lectures look at important economic models that also illustrate common equilibrium concepts.

### 4.1 Schellings Segregation Model

**Contents**

- Schellings Segregation Model
  - Outline
  - The Model
  - Results
  - Exercises
  - Solutions

**4.1.1 Outline**

In 1969, Thomas C. Schelling developed a simple but striking model of racial segregation [Sch69].

His model studies the dynamics of racially mixed neighborhoods.

Like much of Schellings work, the model shows how local interactions can lead to surprising aggregate structure.

In particular, it shows that relatively mild preference for neighbors of similar race can lead in aggregate to the collapse of mixed neighborhoods, and high levels of segregation.

In recognition of this and other research, Schelling was awarded the 2005 Nobel Prize in Economic Sciences (joint with Robert Aumann).

In this lecture we (in fact you) will build and run a version of Schellings model.
4.1.2 The Model

We will cover a variation of Schellings model that is easy to program and captures the main idea

Set Up

Suppose we have two types of people: orange people and green people
For the purpose of this lecture, we will assume there are 250 of each type
These agents all live on a single unit square
The location of an agent is just a point \((x, y)\), where \(0 < x, y < 1\)

Preferences

We will say that an agent is happy if half or more of her 10 nearest neighbors are of the same type
Here nearest is in terms of Euclidean distance
An agent who is not happy is called unhappy
An important point here is that agents are not averse to living in mixed areas
They are perfectly happy if half their neighbors are of the other color

Behavior

Initially, agents are mixed together (integrated)
In particular, the initial location of each agent is an independent draw from a bivariate uniform distribution on \(S = (0, 1)^2\)
Now, cycling through the set of all agents, each agent is now given the chance to stay or move
We assume that each agent will stay put if they are happy and move if unhappy
The algorithm for moving is as follows
1. Draw a random location in \(S\)
2. If happy at new location, move there
3. Else, go to step 1
In this way, we cycle continuously through the agents, moving as required
We continue to cycle until no one wishes to move
4.1.3 Results

Let's have a look at the results we got when we coded and ran this model.

As discussed above, agents are initially mixed randomly together.

But after several cycles they become segregated into distinct regions.
4.1. Schellings Segregation Model
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.
4.1.4 Exercises

Exercise 1

Implement and run this simulation for yourself

Use 250 agents of each type

4.1.5 Solutions

Exercise 1

Here is one solution that does the job we want. If you feel like a further exercise you can probably speed up some of the computations and then increase the number of agents.

```julia
using Plots
pyplot() # set seed for random numbers. Reproducible output

mutable struct Agent{<:Integer, <:AbstractFloat}
    kind::<:Integer
    location::Vector{<:AbstractFloat}
end

# constructor
Agent(k::Integer) = Agent(k, rand(2))

function draw_location!(a::Agent)
    a.location = rand(2)
    nothing
end

# distance is just 2 norm: uses our subtraction function
get_distance(a::Agent, o::Agent) = norm(a.location - o.location)

function is_happy(a::Agent, others::Vector{Agent})
    "True if sufficient number of nearest neighbors are of the same type."
    # distances is a list of pairs (d, agent), where d is distance from
    # agent to self
    distances = Any[]

    for agent in others
        if a != agent
            dist = get_distance(a, agent)
            push!(distances, (dist, agent))
        end
    end

    # == Sort from smallest to largest, according to distance == #
```
sort!(distances)

# == Extract the neighboring agents == #
neighbors = [agent for (d, agent) in distances[1:num_neighbors]]

# == Count how many neighbors have the same type as self == #
num_same_type = sum([a.kind == other.kind for other in neighbors])

return num_same_type >= require_same_type
end

function update!(a::Agent, others::Vector{Agent})
    "If not happy, then randomly choose new locations until happy."
    while !is_happy(a, others)
        draw_location!(a)
    end
    return nothing
end

function plot_distribution(agents::Vector{Agent}, cycle_num)
    x_vals_0, y_vals_0 = Float64[], Float64[]
    x_vals_1, y_vals_1 = Float64[], Float64[]

    # == Obtain locations of each type == #
    for agent in agents
        x, y = agent.location
        if agent.kind == 0
            push!(x_vals_0, x)
            push!(y_vals_0, y)
        else
            push!(x_vals_1, x)
            push!(y_vals_1, y)
        end
    end

    p = scatter(x_vals_0, y_vals_0, color=:orange, markersize=8, alpha=0.6)
    scatter!(x_vals_1, y_vals_1, color=:green, markersize=8, alpha=0.6)
    plot!(title="Cycle $(cycle_num)", legend=:none)

    return p
end

# == Main == #

num_of_type_0 = 250
num_of_type_1 = 250
num_neighbors = 10    # Number of agents regarded as neighbors
require_same_type = 5    # Want at least this many neighbors to be same type

# == Create a list of agents == #
agents = Agent[Agent(0) for i in 1:num_of_type_0]
push!(agents, [Agent(1) for i in 1:num_of_type_1]...)
count = 1

# == Loop until none wishes to move == #
while true
    println("Entering loop $count")
    p = plot_distribution(agents, count)
    display(p)
    count += 1
    no_one_moved = true
    movers = 0
    for agent in agents
        old_location = agent.location
        update!(agent, agents)
        if !isapprox(0.0, maximum(old_location - agent.location))
            no_one_moved = false
        end
    end
    if no_one_moved
        break
    end
end
println("Converged, terminating")

Entering loop 1

Cycle 1
Entering loop 2

Cycle 2

Entering loop 3
4.1. Schellings Segregation Model

Entering loop 4
Converged, terminating
4.2 A Lake Model of Employment and Unemployment

4.2.1 Overview

This lecture describes what has come to be called a lake model. The lake model is a basic tool for modeling unemployment. It allows us to analyze:

- flows between unemployment and employment
- how these flows influence steady state employment and unemployment rates

It is a good model for interpreting monthly labor department reports on gross and net jobs created and jobs destroyed.

The lakes in the model are the pools of employed and unemployed.

The flows between the lakes are caused by:

- firing and hiring
- entry and exit from the labor force

For the first part of this lecture, the parameters governing transitions into and out of unemployment and employment are exogenous.

Later, we will determine some of these transition rates endogenously using the McCall search model.

Well also use some nifty concepts like ergodicity, which provides a fundamental link between cross-sectional and long run time series distributions.

These concepts will help us build an equilibrium model of ex ante homogeneous workers whose different luck generates variations in their ex post experiences.
Prerequisites

Before working through what follows, we recommend you read the lecture on finite Markov chains. You will also need some basic linear algebra and probability.

4.2.2 The Model

The economy is inhabited by a very large number of ex ante identical workers. The workers live forever, spending their lives moving between unemployment and employment. Their rates of transition between employment and unemployment are governed by the following parameters:

- $\lambda$, the job finding rate for currently unemployed workers
- $\alpha$, the dismissal rate for currently employed workers
- $b$, the entry rate into the labor force
- $d$, the exit rate from the labor force

The growth rate of the labor force evidently equals $g = b - d$.

Aggregate Variables

We want to derive the dynamics of the following aggregates:

- $E_t$, the total number of employed workers at date $t$
- $U_t$, the total number of unemployed workers at $t$
- $N_t$, the number of workers in the labor force at $t$

We also want to know the values of the following objects:

- The employment rate $e_t := E_t / N_t$
- The unemployment rate $u_t := U_t / N_t$

(Here and below, capital letters represent stocks and lowercase letters represent flows)

Laws of Motion for Stock Variables

We begin by constructing laws of motion for the aggregate variables $E_t, U_t, N_t$.

Of the mass of workers $E_t$ who are employed at date $t$,

- $(1 - d)E_t$ will remain in the labor force
- of these, $(1 - \alpha)(1 - d)E_t$ will remain employed

Of the mass of workers $U_t$ who are currently unemployed,

- $(1 - d)U_t$ will remain in the labor force
• of these, \((1 - d)\lambda U_t\) will become employed

Therefore, the number of workers who will be employed at date \(t + 1\) will be

\[ E_{t+1} = (1 - d)(1 - \alpha)E_t + (1 - d)\lambda U_t \]

A similar analysis implies

\[ U_{t+1} = (1 - d)\alpha E_t + (1 - d)(1 - \lambda)U_t + b(E_t + U_t) \]

The value \(b(E_t + U_t)\) is the mass of new workers entering the labor force unemployed

The total stock of workers \(N_t = E_t + U_t\) evolves as

\[ N_{t+1} = (1 + b - d)N_t = (1 + g)N_t \]

Letting \(X_t := \begin{pmatrix} U_t \\ E_t \end{pmatrix}\), the law of motion for \(X\) is

\[ X_{t+1} = AX_t \quad \text{where} \quad A := \begin{pmatrix} (1 - d)(1 - \lambda) + b & (1 - d)\alpha + b \\ (1 - d)\lambda & (1 - d)(1 - \alpha) \end{pmatrix} \]

This law tells us how total employment and unemployment evolve over time

**Laws of Motion for Rates**

Now let's derive the law of motion for rates

To get these we can divide both sides of \(X_{t+1} = AX_t\) by \(N_{t+1}\) to get

\[ \begin{pmatrix} U_{t+1}/N_{t+1} \\ E_{t+1}/N_{t+1} \end{pmatrix} = \frac{1}{1 + g} A \begin{pmatrix} U_t/N_t \\ E_t/N_t \end{pmatrix} \]

Letting

\[ x_t := \begin{pmatrix} u_t \\ e_t \end{pmatrix} = \begin{pmatrix} U_t/N_t \\ E_t/N_t \end{pmatrix} \]

we can also write this as

\[ x_{t+1} = \hat{A}x_t \quad \text{where} \quad \hat{A} := \frac{1}{1 + g} A \]

You can check that \(e_t + u_t = 1\) implies that \(e_{t+1} + u_{t+1} = 1\)

This follows from the fact that the columns of \(\hat{A}\) sum to 1

**4.2.3 Implementation**

Let's code up these equations

Here's the code:
#=

@author: Victoria Gregory, John Stachurski

=#

struct LakeModel{TF <: AbstractFloat}
    λ::TF
    α::TF
    b::TF
    d::TF
    g::TF
    A::Matrix{TF}
    A_hat::Matrix{TF}
end

""
Constructor with default values for `LakeModel`

#### Fields of `LakeModel`

- λ : job finding rate
- α : dismissal rate
- b : entry rate into labor force
- d : exit rate from labor force
- g : net entry rate
- A : updates stock
- A_hat : updates rate

""

function LakeModel(λ::AbstractFloat=0.283,
    α::AbstractFloat=0.013,
    b::AbstractFloat=0.0124,
    d::AbstractFloat=0.00822)

    g = b - d
    A = [(1-λ) * (1-d) + b (1-d) * α + b;
         (1-d) * λ   (1-d) * (1-α)]
    A_hat = A ./ (1 + g)

    return LakeModel(λ, α, b, d, g, A, A_hat)
end

""
Finds the steady state of the system \(x_{t+1} = \hat{A} x_t\)

#### Arguments

- im : instance of `LakeModel`
- tol: convergence tolerance

#### Returns

4.2. A Lake Model of Employment and Unemployment
- x : steady state vector of employment and unemployment rates

```
function rate_steady_state(lm::LakeModel, tol::AbstractFloat=1e-6)
    x = 0.5 * ones(2)
    error = tol + 1
    while (error > tol)
        new_x = lm.A_hat * x
        error = maximum(abs, new_x - x)
        x = new_x
    end
    return x
end
```

```
Simulates the the sequence of Employment and Unemployent stocks

```
```
function simulate_stock_path{TF<:AbstractFloat}(lm::LakeModel,
    X0::AbstractVector{TF},
    T::Integer)
    X_path = Array{TF}(2, T)
    X = copy(X0)
    for t in 1:T
        X_path[:, t] = X
        X = lm.A * X
    end
    return X_path
end
```

```
Simulates the the sequence of employment and unemployment rates.

```
```
```
function simulate_rate_path{T<:AbstractFloat}(lm::LakeModel, x0::Vector{T}, T::Integer)

    x_path = Array{T}(2, T)
    x = copy(x0)
    for t in 1:T
        x_path[:, t] = x
        x = lm.A_hat * x
    end
    return x_path
end

lm = LakeModel()

lm.α

0.013

lm.A

2x2 Array{Float64,2}:
  0.723506  0.0252931
  0.280674  0.978887

lm = LakeModel(α = 2.0)
lm.A

2x2 Array{Float64,2}:
  0.723506  1.99596
  0.280674  -0.99178

Aggregate Dynamics

Let’s run a simulation under the default parameters (see above) starting from $X_0 = (12, 138)$

using PyPlot

lm = LakeModel()

N_0 = 150     # Population
e_0 = 0.92    # Initial employment rate
u_0 = 1 - e_0 # Initial unemployment rate
T = 50        # Simulation length

U_0 = u_0 * N_0
E_0 = e_0 * N_0
X_0 = [U_0; E_0]

X_path = simulate_stock_path(lm, X_0, T)
The aggregates $E_t$ and $U_t$ don't converge because their sum $E_t + U_t$ grows at rate $g$

On the other hand, the vector of employment and unemployment rates $x_t$ can be in a steady state $\bar{x}$ if there exists an $\bar{x}$ such that
• $\bar{x} = \hat{A}\bar{x}$

• The components satisfy $\bar{e} + \bar{u} = 1$

This equation tells us that a steady state level $\bar{x}$ is an eigenvector of $\hat{A}$ associated with a unit eigenvalue.

We also have $x_t \to \bar{x}$ as $t \to \infty$ provided that the remaining eigenvalue of $\hat{A}$ has modulus less than 1.

This is the case for our default parameters:

```julia
lm = LakeModel()
e, f = eigvals(lm.A_hat)
abs(e), abs(f)
```

(0.6953067378358462, 1.0)

Let's look at the convergence of the unemployment and employment rate to steady state levels (dashed red line):

```julia
lm = LakeModel()
e_0 = 0.92 # Initial employment rate
u_0 = 1 - e_0 # Initial unemployment rate
T = 50 # Simulation length
xbar = rate_steady_state(lm)
x_0 = [u_0; e_0]
x_path = simulate_rate_path(lm, x_0, T)
titles = [”Unemployment rate” ”Employment rate”]
fig, axes = subplots(2, 1, figsize=(10, 8))
for (i, ax) in enumerate(axes)
    ax[:plot](1:T, x_path[i, :], c=’blue’, lw=2, alpha=0.5)
    ax[:hlines](xbar[i], 0, T, "r", "--")
    ax[:set](title=titles[i])
    ax[:grid]("on")
end
```
4.2.4 Dynamics of an Individual Worker

An individual worker's employment dynamics are governed by a finite state Markov process. The worker can be in one of two states:

- \( s_t = 0 \) means unemployed
- \( s_t = 1 \) means employed

Let's start off under the assumption that \( b = d = 0 \)

The associated transition matrix is then

\[
P = \begin{pmatrix}
1 - \lambda & \lambda \\
\alpha & 1 - \alpha
\end{pmatrix}
\]

Let \( \psi_t \) denote the marginal distribution over employment / unemployment states for the worker at time \( t \). As usual, we regard it as a row vector.

We know from an earlier discussion that \( \psi_t \) follows the law of motion

\[
\psi_{t+1} = \psi_t P
\]
We also know from the *lecture on finite Markov chains* that if \( \alpha \in (0, 1) \) and \( \lambda \in (0, 1) \), then \( P \) has a unique stationary distribution, denoted here by \( \psi^* \).

The unique stationary distribution satisfies

\[
\psi^*[0] = \frac{\alpha}{\alpha + \lambda}
\]

Not surprisingly, probability mass on the unemployment state increases with the dismissal rate and falls with the job finding rate rate.

**Ergodicity**

Let's look at a typical lifetime of employment-unemployment spells.

We want to compute the average amounts of time an infinitely lived worker would spend employed and unemployed.

Let

\[
\bar{s}_{u,T} := \frac{1}{T} \sum_{t=1}^{T} \mathbf{1}\{s_t = 0\}
\]

and

\[
\bar{s}_{e,T} := \frac{1}{T} \sum_{t=1}^{T} \mathbf{1}\{s_t = 1\}
\]

(As usual, \( \mathbf{1}\{Q\} = 1 \) if statement \( Q \) is true and 0 otherwise)

These are the fraction of time a worker spends unemployed and employed, respectively, up until period \( T \).

If \( \alpha \in (0, 1) \) and \( \lambda \in (0, 1) \), then \( P \) is *ergodic*, and hence we have

\[
\lim_{T \to \infty} \bar{s}_{u,T} = \psi^*[0] \quad \text{and} \quad \lim_{T \to \infty} \bar{s}_{e,T} = \psi^*[1]
\]

with probability one.

Inspection tells us that \( P \) is exactly the transpose of \( \hat{A} \) under the assumption \( b = d = 0 \).

Thus, the percentages of time that an infinitely lived worker spends employed and unemployed equal the fractions of workers employed and unemployed in the steady state distribution.

**Convergence rate**

How long does it take for time series sample averages to converge to cross sectional averages?

We can use *QuantEcon.jl's MarkovChain* type to investigate this.

Let's plot the path of the sample averages over 5,000 periods.
using QuantEcon
srand(42)
lm = LakeModel(d=0.0, b=0.0)
T = 5000  # Simulation length
α, λ = lm.α, lm.λ
P = [(1 - λ) λ; α (1 - α)]
mc = MarkovChain(P, [0; 1])  # 0=unemployed, 1=employed
xbar = rate_steady_state(lm)
s_path = simulate(mc, T; init=2)
s_bar_e = cumsum(s_path) ./ (1:T)
s_bar_u = 1 - s_bar_e
s_bars = [s_bar_u s_bar_e]
titles = ["Percent of time unemployed" "Percent of time employed"]
fig, axes = subplots(2, 1, figsize=(10, 8))
for (i, ax) in enumerate(axes)
    ax[:plot](1:T, s_bars[:, i], c="blue", lw=2, alpha=0.5)
    ax[:hlines](xbar[i], 0, T, "r", "--")
    ax[:set](title=titles[i])
    ax[:grid]("on")
end
The stationary probabilities are given by the dashed red line.

In this case it takes much of the sample for these two objects to converge.

This is largely due to the high persistence in the Markov chain.

### 4.2.5 Endogenous Job Finding Rate

We now make the hiring rate endogenous.

The transition rate from unemployment to employment will be determined by the McCall search model \[McC70\].

All details relevant to the following discussion can be found in our treatment of that model.

#### Reservation Wage

The most important thing to remember about the model is that optimal decisions are characterized by a reservation wage \( \tilde{w} \).
• If the wage offer \( w \) in hand is greater than or equal to \( \bar{w} \), then the worker accepts
• Otherwise, the worker rejects

As we saw in our discussion of the model, the reservation wage depends on the wage offer distribution and the parameters
• \( \alpha \), the separation rate
• \( \beta \), the discount factor
• \( \gamma \), the offer arrival rate
• \( c \), unemployment compensation

### Linking the McCall Search Model to the Lake Model

Suppose that all workers inside a lake model behave according to the McCall search model.
The exogenous probability of leaving employment remains \( \alpha \).
But their optimal decision rules determine the probability \( \lambda \) of leaving unemployment.
This is now

\[
\lambda = \gamma \mathbb{P}\{w_t \geq \bar{w}\} = \gamma \sum_{w' \geq \bar{w}} p(w')
\]

### Fiscal Policy

We can use the McCall search version of the Lake Model to find an optimal level of unemployment insurance.
We assume that the government sets unemployment compensation \( c \).
The government imposes a lump sum tax \( \tau \) sufficient to finance total unemployment payments.
To attain a balanced budget at a steady state, taxes, the steady state unemployment rate \( u \), and the unemployment compensation rate must satisfy

\[
\tau = uc
\]
The lump sum tax applies to everyone, including unemployed workers.
Thus, the post-tax income of an employed worker with wage \( w \) is \( w - \tau \).
The post-tax income of an unemployed worker is \( c - \tau \).
For each specification \((c, \tau)\) of government policy, we can solve for the workers optimal reservation wage.
This determines \( \lambda \) via (4.1) evaluated at post tax wages, which in turn determines a steady state unemployment rate \( u(c, \tau) \).
For a given level of unemployment benefit $c$, we can solve for a tax that balances the budget in the steady state

$$\tau = u(c, \tau)c$$

To evaluate alternative government tax-unemployment compensation pairs, we require a welfare criterion. We use a steady state welfare criterion

$$W := e E[V | \text{employed}] + u U$$

where the notation $V$ and $U$ is as defined in the McCall search model lecture.

The wage offer distribution will be a discretized version of the lognormal distribution $LN(\log(20), 1)$, as shown in the next figure.

![Distribution of pre-tax wages](image)

We take a period to be a month.

We set $b$ and $d$ to match monthly birth and death rates, respectively, in the U.S. population:

- $b = 0.0124$
- $d = 0.00822$

Following [DFH06], we set $\alpha$, the hazard rate of leaving employment, to

- $\alpha = 0.013$

**Fiscal Policy Code**

We will make use of code we wrote in the McCall model lecture, embedded below for convenience.
The first piece of code, repeated below, implements value function iteration

```
using Distributions

# A default utility function

function u(c::Real, σ::Real)
    if c > 0
        return (c^(1 - σ) - 1) / (1 - σ)
    else
        return -10e6
    end
end

# default wage vector with probabilities

const n = 60  # n possible outcomes for wage
const default_w_vec = linspace(10, 20, n)  # wages between 10 and 20
const a, b = 600, 400  # shape parameters
const dist = BetaBinomial(n-1, a, b)
const default_p_vec = pdf.(dist, support(dist))

mutable struct McCallModel{TF <: AbstractFloat, TAV <: AbstractVector{TF}, TAV2 <: AbstractVector{TF}}
    α::TF  # Job separation rate
    β::TF  # Discount rate
    γ::TF  # Job offer rate
    c::TF  # Unemployment compensation
    σ::TF  # Utility parameter
    w_vec::TAV  # Possible wage values
    p_vec::TAV2  # Probabilities over w_vec
end

McCallModel(α=0.2, β=0.98, γ=0.7, c=6.0, σ=2.0, w_vec=default_w_vec, p_vec=default_p_vec) where {TF, TAV, TAV2} =
    new(TF, TAV, TAV2)(α, β, γ, c, σ, w_vec, p_vec)

# A function to update the Bellman equations. Note that V_new is modified in place (i.e., modified by this function). The new value of U is returned.

function update_bellman!(mcm::McCallModel, V::AbstractVector, V_new::AbstractVector, U::Real)
    # Simplify notation
    α, β, σ, c, γ = mcm.α, mcm.β, mcm.σ, mcm.c, mcm.γ
```
for (w_idx, w) in enumerate(mcm.w_vec)
    # w_idx indexes the vector of possible wages
    V_new[w_idx] = u(w, σ) + β * ((1 - α) * V[w_idx] + α * U)
end

U_new = u(c, σ) + β * (1 - γ) * U +
    β * γ * dot(max.(U, V), mcm.p_vec)

return U_new
end

function solve_mccall_model(mcm::McCallModel;
    tol::AbstractFloat=1e-5,
    max_iter::Integer=2000)

    V = ones(length(mcm.w_vec))  # Initial guess of V
    V_new = similar(V)  # To store updates to V
    U = 1.0  # Initial guess of U
    i = 0
    error = tol + 1
    while error > tol & & i < max_iter
        U_new = update_bellman!(mcm, V, V_new, U)
        error_1 = maximum(abs, V_new - V)
        error_2 = abs(U_new - U)
        error = max(error_1, error_2)
        V[:] = V_new
        U = U_new
        i += 1
    end

    return V, U
end

The second piece of code repeated from the McCall model lecture is used to complete the reservation wage

###

Computes the reservation wage of an instance of the McCall model by finding the smallest w such that V(w) > U.

If V(w) > U for all w, then the reservation wage w_bar is set to the lowest wage in mcm.w_vec.

If v(w) < U for all w, then w_bar is set to np.inf.

Parameters
---------

mcm : an instance of McCallModel

return_values : bool (optional, default=false)

Return the value functions as well.
Now lets compute and plot welfare, employment, unemployment, and tax revenue as a function of the unemployment compensation rate

```julia
# Some global variables that will stay constant
α = 0.013
α_q = (1 - (1 - α)^3)
b_param = 0.0124
d_param = 0.00822
β = 0.98
γ = 1.0
σ = 2.0

# The default wage distribution: a discretized log normal
log_wage_mean, wage_grid_size, max_wage = 20, 200, 170
w_vec = linspace(1e-3, max_wage, wage_grid_size + 1)
logw_dist = Normal(log(log_wage_mean), 1)
cdf_logw = cdf_logw[2:end] - cdf_logw[1:end-1]
p_vec = pdf_logw ./ sum(pdf_logw)
w_vec = (w_vec[1:end-1] + w_vec[2:end]) / 2

Compute the reservation wage, job finding rate and value functions of the workers given c and τ.

returns
------
w_bar : scalar
    The reservation wage

function compute_reservation_wage(mcm::McCallModel; return_values::Bool=false)
    V, U = solve_mccall_model(mcm)
w_idx = searchsortedfirst(V - U, 0)
    if w_idx == length(V)
        w_bar = Inf
    else
        w_bar = mcm.w_vec[w_idx]
    end
    if return_values == false
        return w_bar
    else
        return w_bar, V, U
    end
end
```

Chapter 4. Multiple Agent Models
\[\beta,\gamma, c-\tau, \sigma, \text{collect}(w_vec-\tau), \text{post-tax compensation}\]

\[p_vec, \text{post-tax wages}\]

\[
w_bar, V, U = \text{compute_reservation_wage}(\text{mcm}, \text{return_values=}true)\]
\[
\lambda = \gamma \ast \text{sum}(p_vec[w_vec-\tau .> w_bar])\]

\[
\text{return } w_bar, \lambda, V, U\]

""
Compute the steady state unemployment rate given \(c\) and \(\tau\) using optimal quantities from the McCall model and computing corresponding steady state quantities
""

\[
\text{function compute_steady_state_quantities}(c::\text{AbstractFloat}, \tau::\text{AbstractFloat})\]
\[
w_bar, \lambda\_\text{param}, V, U = \text{compute_optimal_quantities}(c, \tau)\]

# Compute steady state employment and unemployment rates
\[
\text{lm} = \text{LakeModel}(\lambda=\lambda\_\text{param}, \alpha=\alpha_q, b=b\_\text{param}, d=d\_\text{param})\]
\[
x = \text{rate_steady_state}(\text{lm})\]
\[
u\_\text{rate}, e\_\text{rate} = x\]

# Compute steady state welfare
\[
w = \text{sum}(V .* p_vec .* (w_vec-\tau .> w_bar)) / \text{sum}(p_vec .* (w_vec-\tau .> u->w_bar))\]
\[
\text{welfare} = e\_\text{rate} .* w + u\_\text{rate} .* U\]

\[
\text{return } u\_\text{rate}, e\_\text{rate}, \text{welfare}\]

""
Find tax level that will induce a balanced budget.
""

\[
\text{function find_balanced_budget_tax}(c::\text{Real})\]
\[
\text{function steady_state_budget}(t::\text{Real})\]
\[
\text{u\_rate, e\_rate, w} = \text{compute_steady_state_quantities}(c, t)\]
\[
\text{return } t - u\_\text{rate} \ast c\]
\[
\text{end}\]
\[
\tau = \text{brent}(\text{steady_state_budget}, 0.0, 0.9 \ast c)\]
\[
\text{return } \tau\]
\[
\text{end}\]

# Levels of unemployment insurance we wish to study
\[
Nc = 60\]
The figure that the preceding code listing generates is shown below.
Welfare first increases and then decreases as unemployment benefits rise
The level that maximizes steady state welfare is approximately 62

4.2.6 Exercises

Exercise 1

Consider an economy with initial stock of workers $N_0 = 100$ at the steady state level of employment in the baseline parameterization

- $\alpha = 0.013$
- $\lambda = 0.283$
- $b = 0.0124$
- $d = 0.00822$
(The values for \( \alpha \) and \( \lambda \) follow [DFH06])

Suppose that in response to new legislation the hiring rate reduces to \( \lambda = 0.2 \)

Plot the transition dynamics of the unemployment and employment stocks for 50 periods

Plot the transition dynamics for the rates

How long does the economy take to converge to its new steady state?

What is the new steady state level of employment?

**Exercise 2**

Consider an economy with initial stock of workers \( N_0 = 100 \) at the steady state level of employment in the baseline parameterization

Suppose that for 20 periods the birth rate was temporarily high (\( b = 0.0025 \)) and then returned to its original level

Plot the transition dynamics of the unemployment and employment stocks for 50 periods

Plot the transition dynamics for the rates

How long does the economy take to return to its original steady state?

### 4.2.7 Solutions

**Exercise 1**

We begin by constructing the type containing the default parameters and assigning the steady state values to \( x_0 \)

```julia
lm = LakeModel()
x0 = rate_steady_state(lm)
println("Initial Steady State: \\
\$x0\"")
```

Initial Steady State: [0.0826681, 0.917332]

Initialize the simulation values

```julia
N0 = 100
T = 50
```

New legislation changes \( \lambda \) to 0.2

```julia
lm = LakeModel(\lambda=0.2)
```

LakeModel{Float64}(0.2, 0.013, 0.0124, 0.00822, 0.00418, [0.805824 0.0252931; 0.198356 0.978887], [0.80247 0.0251879; 0.19753 0.974812])
xbar = rate_steady_state(lm) # new steady state
X_path = simulate_stock_path(lm,x0 * N0, T)
x_path = simulate_rate_path(lm,x0, T)
println("New Steady State: \$xbar")

New Steady State: [0.113096, 0.886904]

Now plot stocks

titles = ["Unemployment" "Employment" "Labor force"]

x1 = X_path[1, :]
x2 = X_path[2, :]
x3 = squeeze(sum(X_path, 1), 1)
fig, axes = subplots(3, 1, figsize=(10, 8))

for (ax, x, title) in zip(axes, [x1, x2, x3], titles)
    ax[:plot](1:T, x, c="blue")
    ax[:set](title=title)
    ax[:grid]("on")
end

fig[:tight_layout]()}
And how the rates evolve

```julia
titles = ["Unemployment rate" "Employment rate"]
fig, axes = subplots(2, 1, figsize=(10, 8))
for (i, ax) in enumerate(axes)
    ax[:plot](1:T, x_path[i, :], c="blue", lw=2, alpha=0.5)
    ax[:hlines](xbar[i], 0, T, "r", "--")
    ax[:set](title=titles[i])
    ax[:grid]("on")
end
```
We see that it takes 20 periods for the economy to converge to its new steady state levels

**Exercise 2**

This next exercise has the economy experiencing a boom in entrances to the labor market and then later returning to the original levels.

For 20 periods the economy has a new entry rate into the labor market.

Let's start off at the baseline parameterization and record the steady state levels.

```julia
lm = LakeModel()
x0 = rate_steady_state(lm)
```

2-element Array{Float64,1}:
0.0826681
0.917332

Here are the other parameters:

```julia
b_hat = 0.003
T_hat = 20
```
Now we reset \( b \) to the original value and then, using the state after 20 periods for the new initial conditions, we simulate for the additional 30 periods.

```julia
lm = LakeModel(b=0.0124)
X_path2 = simulate_stock_path(lm, X_path1[:, end-1], T-T_hat+1)  # simulate stocks
x_path2 = simulate_rate_path(lm, x_path1[:, end-1], T-T_hat+1)  # simulate rates

2×31 Array{Float64,2}:
0.0536401 0.0624842 0.0686335 0.0826652 0.0826655 0.0826657
0.94636 0.937516 0.931366 0.917335 0.917335 0.917334
```

Finally we combine these two paths and plot.

```julia
x_path = hcat(x_path1, x_path2[:, 2:end])  # note [2:] to avoid doubling period 20
X_path = hcat(X_path1, X_path2[:, 2:end])

2×50 Array{Float64,2}:
91.7332  92.1167  92.2379  92.1769  93.8293  94.2215  94.6153
```

```julia
titles = ["Unemployment" "Employment" "Labor force"]
x1 = X_path[1,:]
x2 = X_path[2,:]
x3 = squeeze(sum(X_path, 1), 1)
fig, axes = subplots(3, 1, figsize=(10, 9))

for (ax, x, title) in zip(axes, [x1, x2, x3], titles)
    ax[:plot](1:T, x, "b-", lw=2, alpha=0.7)
    ax[:set](title=title, ylim=(minimum(x-1), maximum(x+1)))
    ax[:grid]("on")
end
fig[:tight_layout]()
```
And the rates

titles = ["Unemployment Rate" "Employment Rate"]

fig, axes = subplots(2, 1, figsize=(10, 8))

for (i, ax) in enumerate(axes)
    ax[:plot](1:T, x_path[i, :], "-b", lw=2, alpha=0.7)
    ax[:hlines](x0[i], 0, T, "r", "--")
    ax[:set](title=titles[i])
    ax[:grid]("on")
end

4.2. A Lake Model of Employment and Unemployment
4.3 Rational Expectations Equilibrium

Contents

• Rational Expectations Equilibrium
  – Overview
  – Defining Rational Expectations Equilibrium
  – Computation of an Equilibrium
  – Exercises
  – Solutions

If you're so smart, why aren't you rich?
4.3.1 Overview

This lecture introduces the concept of rational expectations equilibrium.

To illustrate it, we describe a linear quadratic version of a famous and important model due to Lucas and Prescott [LP71].

This 1971 paper is one of a small number of research articles that kicked off the rational expectations revolution.

We follow Lucas and Prescott by employing a setting that is readily Bellmanized (i.e., capable of being formulated in terms of dynamic programming problems).

Because we use linear quadratic setups for demand and costs, we can adapt the LQ programming techniques described in this lecture.

We will learn about how a representative agents problem differs from a planners, and how a planning problem can be used to compute rational expectations quantities.

We will also learn about how a rational expectations equilibrium can be characterized as a fixed point of a mapping from a perceived law of motion to an actual law of motion.

Equality between a perceived and an actual law of motion for endogenous market-wide objects captures in a nutshell what the rational expectations equilibrium concept is all about.

Finally, we will learn about the important Big $K$, little $k$ trick, a modeling device widely used in macroeconomics.

Except that for us

- Instead of Big $K$ it will be Big $Y$
- Instead of little $k$ it will be little $y$

The Big $Y$, little $y$ trick

This widely used method applies in contexts in which a representative firm or agent is a price taker operating within a competitive equilibrium.

We want to impose that

- The representative firm or individual takes aggregate $Y$ as given when it chooses individual $y$, but . . .
- At the end of the day, $Y = y$, so that the representative firm is indeed representative.

The Big $Y$, little $y$ trick accomplishes these two goals by

- Taking $Y$ as beyond control when posing the choice problem of who chooses $y$; but . . .
- Imposing $Y = y$ after having solved the individuals optimization problem.

Please watch for how this strategy is applied as the lecture unfolds.

We begin by applying the Big $Y$, little $y$ trick in a very simple static context.
A simple static example of the Big $Y$, little $y$ trick

Consider a static model in which a collection of $n$ firms produce a homogeneous good that is sold in a competitive market.

Each of these $n$ firms sells output $y$.

The price $p$ of the good lies on an inverse demand curve

$$ p = a_0 - a_1 Y $$ \hspace{1cm} (4.2)

where

- $a_i > 0$ for $i = 0, 1$
- $Y = ny$ is the market-wide level of output

Each firm has total cost function

$$ c(y) = c_1 y + 0.5c_2 y^2, \quad c_i > 0 \text{ for } i = 1, 2 $$

The profits of a representative firm are $py - c(y)$

Using (4.2), we can express the problem of the representative firm as

$$ \max_y \left[ (a_0 - a_1 Y) y - c_1 y - 0.5c_2 y^2 \right] $$ \hspace{1cm} (4.3)

In posing problem (4.3), we want the firm to be a price taker.

We do that by regarding $p$ and therefore $Y$ as exogenous to the firm.

The essence of the Big $Y$, little $y$ trick is not to set $Y = ny$ before taking the first-order condition with respect to $y$ in problem (4.3).

This assures that the firm is a price taker.

The first order condition for problem (4.3) is

$$ a_0 - a_1 Y - c_1 - c_2 y = 0 $$ \hspace{1cm} (4.4)

At this point, but not before, we substitute $Y = ny$ into (4.4) to obtain the following linear equation

$$ a_0 - c_1 - (a_1 + n^{-1} c_2) Y = 0 $$ \hspace{1cm} (4.5)

to be solved for the competitive equilibrium market wide output $Y$.

After solving for $Y$, we can compute the competitive equilibrium price $p$ from the inverse demand curve (4.2).
Further Reading

References for this lecture include

• [LP71]
• [Sar87], chapter XIV
• [LS18], chapter 7

4.3.2 Defining Rational Expectations Equilibrium

Our first illustration of a rational expectations equilibrium involves a market with \( n \) firms, each of which seeks to maximize the discounted present value of profits in the face of adjustment costs.

The adjustment costs induce the firms to make gradual adjustments, which in turn requires consideration of future prices.

Individual firms understand that, via the inverse demand curve, the price is determined by the amounts supplied by other firms.

Hence each firm wants to forecast future total industry supplies.

In our context, a forecast is generated by a belief about the law of motion for the aggregate state.

Rational expectations equilibrium prevails when this belief coincides with the actual law of motion generated by production choices induced by this belief.

We formulate a rational expectations equilibrium in terms of a fixed point of an operator that maps beliefs into optimal beliefs.

Competitive Equilibrium with Adjustment Costs

To illustrate, consider a collection of \( n \) firms producing a homogeneous good that is sold in a competitive market.

Each of these \( n \) firms sells output \( y_t \).

The price \( p_t \) of the good lies on the inverse demand curve

\[
p_t = a_0 - a_1 Y_t
\]

(4.6)

where

• \( a_i > 0 \) for \( i = 0, 1 \)
• \( Y_t = ny_t \) is the market-wide level of output
The Firms Problem

Each firm is a price taker
While it faces no uncertainty, it does face adjustment costs
In particular, it chooses a production plan to maximize

$$\sum_{t=0}^{\infty} \beta^t r_t$$  \hspace{1cm} (4.7)

where

$$r_t := p_t y_t - \gamma \frac{(y_{t+1} - y_t)^2}{2}, \quad y_0 \text{ given}$$  \hspace{1cm} (4.8)

Regarding the parameters,

• $\beta \in (0, 1)$ is a discount factor

• $\gamma > 0$ measures the cost of adjusting the rate of output

Regarding timing, the firm observes $p_t$ and $y_t$ when it chooses $y_{t+1}$ at time $t$
To state the firms optimization problem completely requires that we specify dynamics for all state variables
This includes ones that the firm cares about but does not control like $p_t$
We turn to this problem now

Prices and Aggregate Output

In view of (4.6), the firms incentive to forecast the market price translates into an incentive to forecast aggregate output $Y_t$

Aggregate output depends on the choices of other firms
We assume that $n$ is such a large number that the output of any single firm has a negligible effect on aggregate output
That justifies firms in regarding their forecasts of aggregate output as being unaffected by their own output decisions

The Firms Beliefs

We suppose the firm believes that market-wide output $Y_t$ follows the law of motion

$$Y_{t+1} = H(Y_t)$$  \hspace{1cm} (4.9)
where \( Y_0 \) is a known initial condition

The belief function \( H \) is an equilibrium object, and hence remains to be determined

**Optimal Behavior Given Beliefs**

For now let’s fix a particular belief \( H \) in (4.9) 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

\[
v(y, Y) = \max_{y'} \left\{ a_0 y - a_1 y Y + \frac{\gamma (y' - y)^2}{2} + \beta v(y', H(Y)) \right\}
\]  

(4.10)

Let’s denote the firm’s optimal policy function by \( h \), so that

\[
y_{t+1} = h(y_t, Y_t)
\]  

(4.11)

where

\[
h(y, Y) := \arg \max_{y'} \left\{ a_0 y - a_1 y Y + \frac{\gamma (y' - y)^2}{2} + \beta v(y', H(Y)) \right\}
\]  

(4.12)

Evidently \( v \) and \( h \) both depend on \( H \).

**First-Order Characterization of \( h \)**

In what follows it will be helpful to have a second characterization of \( h \), based on first order conditions.

The first-order necessary condition for choosing \( y' \) is

\[
-\gamma (y' - y) + \beta v_y(y', H(Y)) = 0
\]  

(4.13)

An important useful envelope result of Benveniste-Scheinkman [BS79] implies that to differentiate \( v \) with respect to \( y \) we can naively differentiate the right side of (4.10), giving

\[
v_y(y, Y) = a_0 - a_1 Y + \gamma (y' - y)
\]

Substituting this equation into (4.13) gives the Euler equation

\[
-\gamma (y_{t+1} - y_t) + \beta [a_0 - a_1 Y_{t+1} + \gamma (y_{t+2} - y_{t+1})] = 0
\]  

(4.14)

The firm optimally sets an output path that satisfies (4.14), taking (4.9) as given, and subject to

4.3. Rational Expectations Equilibrium
• the initial conditions for \((y_0, Y_0)\)
• the terminal condition \(\lim_{t \to \infty} \beta^t y_t v_t(y_t, Y_t) = 0\)

This last condition is called the **transversality condition**, and acts as a first-order necessary condition at infinity.

The firms decision rule solves the difference equation (4.14) subject to the given initial condition \(y_0\) and the transversality condition.

Note that solving the Bellman equation (4.10) for \(v\) and then \(h\) in (4.12) yields a decision rule that automatically imposes both the Euler equation (4.14) and the transversality condition.

### The Actual Law of Motion for \(\{Y_t\}\)

As we've seen, a given belief translates into a particular decision rule \(h\).

Recalling that \(Y_t = ny_t\), the actual law of motion for market-wide output is then

\[
Y_{t+1} = nh(Y_t/n, Y_t)
\]  

(4.15)

Thus, when firms believe that the law of motion for market-wide output is (4.9), their optimizing behavior makes the actual law of motion be (4.15).

### Definition of Rational Expectations Equilibrium

A **rational expectations equilibrium** or recursive competitive equilibrium of the model with adjustment costs is a decision rule \(h\) and an aggregate law of motion \(H\) such that

1. Given belief \(H\), the map \(h\) is the firms optimal policy function.
2. The law of motion \(H\) satisfies \(H(Y) = nh(Y/n, Y)\) for all \(Y\).

Thus, a rational expectations equilibrium equates the perceived and actual laws of motion (4.9) and (4.15).

### Fixed point characterization

As we've seen, the firms optimum problem induces a mapping \(\Phi\) from a perceived law of motion \(H\) for market-wide output to an actual law of motion \(\Phi(H)\).

The mapping \(\Phi\) is the composition of two operations, taking a perceived law of motion into a decision rule via (4.10)–(4.12), and a decision rule into an actual law via (4.15).

The \(H\) component of a rational expectations equilibrium is a fixed point of \(\Phi\).

### 4.3.3 Computation of an Equilibrium

Now let's consider the problem of computing the rational expectations equilibrium.
**Misbehavior of \( \Phi \)**

Readers accustomed to dynamic programming arguments might try to address this problem by choosing some guess \( H_0 \) for the aggregate law of motion and then iterating with \( \Phi \).

Unfortunately, the mapping \( \Phi \) is not a contraction.

In particular, there is no guarantee that direct iterations on \( \Phi \) converge\(^1\).

Fortunately, there is another method that works here.

The method exploits a general connection between equilibrium and Pareto optimality expressed in the fundamental theorems of welfare economics (see, e.g., [MCWG95]).

Lucas and Prescott [LP71] used this method to construct a rational expectations equilibrium.

The details follow.

**A Planning Problem Approach**

Our plan of attack is to match the Euler equations of the market problem with those for a single-agent choice problem.

As well see, this planning problem can be solved by LQ control (linear regulator).

The optimal quantities from the planning problem are rational expectations equilibrium quantities.

The rational expectations equilibrium price can be obtained as a shadow price in the planning problem.

For convenience, in this section we set \( n = 1 \).

We first compute a sum of consumer and producer surplus at time \( t \):

\[
s(Y_t, Y_{t+1}) := \int_0^{Y_t} (a_0 - a_1 x) \, dx - \frac{\gamma (Y_{t+1} - Y_t)^2}{2}
\]

(4.16)

The first term is the area under the demand curve, while the second measures the social costs of changing output.

The *planning problem* is to choose a production plan \( \{Y_t\} \) to maximize

\[
\sum_{t=0}^{\infty} \beta^t s(Y_t, Y_{t+1})
\]

subject to an initial condition for \( Y_0 \).

---

\(^1\) A literature that studies whether models populated with agents who learn can converge to rational expectations equilibria features iterations on a modification of the mapping \( \Phi \) 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.
**Solution of the Planning Problem**

Evaluating the integral in (4.16) yields the quadratic form $a_0 Y_t - a_1 Y_t^2 / 2$

As a result, the Bellman equation for the planning problem is

$$V(Y) = \max_{Y'} \left\{ a_0 Y - \frac{a_1}{2} Y^2 - \frac{\gamma(Y' - Y)^2}{2} + \beta V(Y') \right\}$$

(4.17)

The associated first order condition is

$$-\gamma(Y' - Y) + \beta V'(Y') = 0$$

(4.18)

Applying the same Benveniste-Scheinkman formula gives

$$V'(Y) = a_0 - a_1 Y + \gamma(Y' - Y)$$

Substituting this into equation (4.18) and rearranging leads to the Euler equation

$$\beta a_0 + \gamma Y_t - [\beta a_1 + \gamma(1 + \beta)] Y_{t+1} + \gamma \beta Y_{t+2} = 0$$

(4.19)

**The Key Insight**

Return to equation (4.14) and set $y_t = Y_t$ for all $t$

(Recall that for this section we set $n = 1$ to simplify the calculations)

A small amount of algebra will convince you that when $y_t = Y_t$, equations (4.19) and (4.14) are identical

Thus, the Euler equation for the planning problem matches the second-order difference equation that we derived by

1. finding the Euler equation of the representative firm and
2. substituting into it the expression $Y_t = n y_t$ that makes the representative firm be representative

If it is appropriate to apply the same terminal conditions for these two difference equations, which it is, then we have verified that a solution of the planning problem is also a rational expectations equilibrium quantity sequence

It follows that for this example we can compute equilibrium quantities by forming the optimal linear regulator problem corresponding to the Bellman equation (4.17)

The optimal policy function for the planning problem is the aggregate law of motion $H$ that the representative firm faces within a rational expectations equilibrium.
Structure of the Law of Motion

As you are asked to show in the exercises, the fact that the planners problem is an LQ problem implies an optimal policy and hence aggregate law of motion taking the form

\[ Y_{t+1} = \kappa_0 + \kappa_1 Y_t \]  

(4.20)

for some parameter pair \( \kappa_0, \kappa_1 \)

Now that we know the aggregate law of motion is linear, we can see from the firms Bellman equation (4.10) that the firms problem can also be framed as an LQ problem

As you are asked to show in the exercises, the LQ formulation of the firms problem implies a law of motion that looks as follows

\[ y_{t+1} = h_0 + h_1 y_t + h_2 Y_t \]  

(4.21)

Hence a rational expectations equilibrium will be defined by the parameters \( (\kappa_0, \kappa_1, h_0, h_1, h_2) \) in (4.20)–(4.21)

4.3.4 Exercises

Exercise 1

Consider the firm problem described above

Let the firms belief function \( H \) be as given in (4.20)

Formulate the firms problem as a discounted optimal linear regulator problem, being careful to describe all of the objects needed

Use the type \( \text{LQ} \) from the \texttt{QuantEcon.jl} package to solve the firms problem for the following parameter values:

\[ a_0 = 100, \ a_1 = 0.05, \ \beta = 0.95, \ \gamma = 10, \ \kappa_0 = 95.5, \ \kappa_1 = 0.95 \]

Express the solution of the firms problem in the form (4.21) and give the values for each \( h_j \)

If there were \( n \) identical competitive firms all behaving according to (4.21), what would (4.21) imply for the actual law of motion (4.9) for market supply

Exercise 2

Consider the following \( \kappa_0, \kappa_1 \) pairs as candidates for the aggregate law of motion component of a rational expectations equilibrium (see (4.20))

Extending the program that you wrote for exercise 1, determine which if any satisfy the definition of a rational expectations equilibrium
Describe an iterative algorithm that uses the program that you wrote for exercise 1 to compute a rational expectations equilibrium 
(You are not being asked actually to use the algorithm you are suggesting)

Exercise 3

Recall the planners problem described above
1. Formulate the planners problem as an LQ problem
2. Solve it using the same parameter values in exercise 1
   • $a_0 = 100, a_1 = 0.05, \beta = 0.95, \gamma = 10$
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 (4.6) and chooses $Y_t$ to maximize $\sum_{t=0}^{\infty} \beta^t r_t$ where

$$r_t = p_t Y_t - \frac{\gamma (Y_{t+1} - Y_t)^2}{2}$$

Formulate this problem as an LQ problem

Compute the optimal policy using the same parameters as the previous exercise

In particular, solve for the parameters in

$$Y_{t+1} = m_0 + m_1 Y_t$$

Compare your results with the previous exercise. Comment.

4.3.5 Solutions

The following solutions were put together by Chase Coleman, Spencer Lyon, Thomas Sargent and John Stachurski

Exercise 1

```
using QuantEcon
```

To map a problem into a discounted optimal linear control problem, we need to define
• state vector $x_t$ and control vector $u_t$

• matrices $A, B, Q, R$ that define preferences and the law of motion for the state

For the state and control vectors we choose

$$x_t = \begin{bmatrix} y_t \\ Y_t \\ 1 \end{bmatrix}, \quad u_t = y_{t+1} - y_t$$

For $A, B, Q, R$ we set

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \kappa_1 & \kappa_0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad R = \begin{bmatrix} 0 & \frac{a_1}{2} & -\frac{a_0}{2} \\ \frac{a_1}{2} & 0 & 0 \\ -\frac{a_0}{2} & 0 & 0 \end{bmatrix}, \quad Q = \gamma/2$$

By multiplying out you can confirm that

• $x'_t Rx_t + u'_t Qu_t = -r_t$

• $x_{t+1} = Ax_t + Bu_t$

Well use the module `lqcontrol.jl` to solve the firm's problem at the stated parameter values

This will return an LQ policy $F$ with the interpretation $u_t = -Fx_t$, or

$$y_{t+1} - y_t = -F_0y_t - F_1Y_t - F_2$$

Matching parameters with $y_{t+1} = h_0 + h_1y_t + h_2Y_t$ leads to

$$h_0 = -F_2, \quad h_1 = 1 - F_0, \quad h_2 = -F_1$$

Here's our solution

```julia
# == Model parameters == #
a0 = 100
a1 = 0.05
\beta = 0.95
\gamma = 10.0

# == Beliefs == #
\kappa0 = 95.5
\kappa1 = 0.95

# == Formulate the LQ problem == #
A = [1 0 0
     0 \kappa1 \kappa0
     0 0 1]
B = [1.0, 0.0, 0.0]
R = [0 a1/2 -a0/2
     a1/2 0 0
     -a0/2 0 0]
```

4.3. Rational Expectations Equilibrium

643
Q = 0.5 * γ

# == Solve for the optimal policy == #
lq = LQ(Q, R, A, B; bet=β)
F, P, d = stationary_values(lq)

hh = h0, h1, h2 = -F[3], 1 - F[1], -F[2]

@printf("F = [%.3f, %.3f, %.3f]\n", F[1], F[2], F[3])
@printf("(h0, h1, h2) = [%.3f, %.3f, %.3f]\n", h0, h1, h2)

F = [-0.000, 0.046, -96.949]
(h0, h1, h2) = [96.949, 1.000, -0.046]

The implication is that

\[ y_{t+1} = 96.949 + y_t - 0.046 Y_t \]

For the case \( n > 1 \), recall that \( Y_t = n y_t \), which, combined with the previous equation, yields

\[ Y_{t+1} = n (96.949 + y_t - 0.046 Y_t) = n96.949 + (1 - n0.046)Y_t \]

**Exercise 2**

To determine whether a \( \kappa_0, \kappa_1 \) pair forms the aggregate law of motion component of a rational expectations equilibrium, we can proceed as follows:

- Determine the corresponding firm law of motion \( y_{t+1} = h_0 + h_1 y_t + h_2 Y_t \)
- Test whether the associated aggregate law \( Y_{t+1} = n h(Y_t / n, Y_t) \) evaluates to \( Y_{t+1} = \kappa_0 + \kappa_1 Y_t \)

In the second step we can use \( Y_t = n y_t = y_t \), so that \( Y_{t+1} = n h(Y_t / n, Y_t) \) becomes

\[ Y_{t+1} = h(Y_t, Y_t) = h_0 + (h_1 + h_2)Y_t \]

Hence to test the second step we can test \( \kappa_0 = h_0 \) and \( \kappa_1 = h_1 + h_2 \)

The following code implements this test

```julia
candidates = ([94.0886298678, 0.923409232937],
               [93.2119845412, 0.984323478873],
               [95.0818452486, 0.952459076301])

for (k0, k1) in candidates
    A = [1 0 0
         0 k1 k0
         0 0 1]
    lq = LQ(Q, R, A, B; bet=β)
    F, P, d = stationary_values(lq)
    hh = h0, h1, h2 = -F[3], 1 - F[1], -F[2]

    if isapprox(k0, h0; atol = 1e-4) && isapprox(k1, h1 + h2; atol = 1e-4)
```
Equilibrium pair = (95.081845, 0.952459)
(h0, h1, h2) = [95.081891, 1.000000, -0.047541]

The output tells us that the answer is pair (iii), which implies $(h_0, h_1, h_2) = (95.0819, 1.0000, -0.0475)$
(Notice we use isapprox to test equality of floating point numbers, since exact equality is too strict)

Regarding the iterative algorithm, one could loop from a given $(\kappa_0, \kappa_1)$ pair to the associated firm law and then to a new $(\kappa_0, \kappa_1)$ pair

This amounts to implementing the operator $\Phi$ described in the lecture

(There is in general no guarantee that this iterative process will converge to a rational expectations equilibrium)

Exercise 3

We are asked to write the planner problem as an LQ problem

For the state and control vectors we choose

$$x_t = \begin{bmatrix} Y_t \\ 1 \end{bmatrix}, \quad u_t = Y_{t+1} - Y_t$$

For the LQ matrices we set

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad R = \begin{bmatrix} a_1/2 & -a_0/2 \\ -a_0/2 & 0 \end{bmatrix}, \quad Q = \gamma/2$$

By multiplying out you can confirm that

- $x_t'Rx_t + u_t'Qu_t = -s(Y_t, Y_{t+1})$
- $x_{t+1} = Ax_t + Bu_t$

By obtaining the optimal policy and using $u_t = -Fx_t$ or

$$Y_{t+1} - Y_t = -F_0Y_t - F_1$$

we can obtain the implied aggregate law of motion via $\kappa_0 = -F_1$ and $\kappa_1 = 1 - F_0$

The Python code to solve this problem is below:

```python
# == Formulate the planner's LQ problem == #
A = eye(2)
B = [1.0, 0.0]
R = [ a1 / 2.0, -a0 / 2.0, -a0 / 2.0, 0.0]
```
Q = γ / 2.0

# == Solve for the optimal policy == #
lq = LQ(Q, R, A, B; bet=β)
P, F, d = stationary_values(lq)

# == Print the results == #
κ0, κ1 = -F[2], 1 - F[1]
println("κ0=$κ0\tκ1=$κ1")

κ0=95.08187459215002  κ1=0.9524590627039248

The output yields the same (κ0, κ1) pair obtained as an equilibrium from the previous exercise

**Exercise 4**

The monopolists LQ problem is almost identical to the planners problem from the previous exercise, except that

\[
R = \begin{bmatrix}
a_1 & -a_0/2 \\
-a_0/2 & 0
\end{bmatrix}
\]

The problem can be solved as follows

# == Formulate the monopolist's LQ problem == #
A = eye(2)
B = [1.0, 0.0]
R = [ a1 -a0 / 2.0
      -a0 / 2.0  0.0]
Q = γ / 2.0

# == Solve for the optimal policy == #
lq = LQ(Q, R, A, B; bet=β)
P, F, d = stationary_values(lq)

# == Print the results == #
m0, m1 = -F[2], 1 - F[1]
println("m0=$m0\tm1=$m1")

m0=73.47294403502833  m1=0.9265270559649701

We see that the law of motion for the monopolist is approximately \( Y_{t+1} = 73.4729 + 0.9265Y_t \)

In the rational expectations case the law of motion was approximately \( Y_{t+1} = 95.0819 + 0.9525Y_t \)

One way to compare these two laws of motion is by their fixed points, which give long run equilibrium output in each case

For laws of the form \( Y_{t+1} = c_0 + c_1Y_t \), the fixed point is \( c_0/(1 - c_1) \)
If you crunch the numbers, you will see that the monopolist adopts a lower long run quantity than obtained by the competitive market, implying a higher market price

This is analogous to the elementary static-case results

4.4 Markov Perfect Equilibrium

4.4.1 Overview

This lecture describes the concept of Markov perfect equilibrium

Markov perfect equilibrium is a key notion for analyzing economic problems involving dynamic strategic interaction, and a cornerstone of applied game theory

In this lecture we teach Markov perfect equilibrium by example

We will focus on settings with

- two players
- quadratic payoff functions
- linear transition rules for the state

Other references include chapter 7 of [LS18]

4.4.2 Background

Markov perfect equilibrium is a refinement of the concept of Nash equilibrium

It is used to study settings where multiple decision makers interact non-cooperatively over time, each seeking to pursue its own objective

The agents in the model face a common state vector, the time path of which is influenced by – and influences – their decisions

In particular, the transition law for the state that confronts each agent is affected by decision rules of other agents

Individual payoff maximization requires that each agent solve a dynamic programming problem that includes this transition law

Markov perfect equilibrium prevails when no agent wishes to revise its policy, taking as given the policies of all other agents

Well known examples include

- Choice of price, output, location or capacity for firms in an industry (e.g., [EP95], [Rya12], [DS10])
- Rate of extraction from a shared natural resource, such as a fishery (e.g., [LM80], [VL11])

Lets examine a model of the first type
**Example: A duopoly model**

Two firms are the only producers of a good the demand for which is governed by a linear inverse demand function

\[
p = a_0 - a_1(q_1 + q_2)
\]

(4.22)

Here \(p = p_t\) is the price of the good, \(q_i = q_{it}\) is the output of firm \(i = 1, 2\) at time \(t\) and \(a_0 > 0, a_1 > 0\).

In (4.22) and what follows,

- the time subscript is suppressed when possible to simplify notation
- \(\hat{x}\) denotes a next period value of variable \(x\)

Each firm recognizes that its output affects total output and therefore the market price.

The one-period payoff function of firm \(i\) is price times quantity minus adjustment costs:

\[
\pi_i = pq_i - \gamma(\hat{q}_i - q_i)^2, \quad \gamma > 0,
\]

(4.23)

Substituting the inverse demand curve (4.22) into (4.23) lets us express the one-period payoff as

\[
\pi_i(q_i, q_{-i}, \hat{q}_i) = a_0q_i - a_1\hat{q}_i^2 - a_1q_iq_{-i} - \gamma(\hat{q}_i - q_i)^2,
\]

(4.24)

where \(q_{-i}\) denotes the output of the firm other than \(i\).

The objective of the firm is to maximize \(\sum_{t=0}^{\infty} \beta^t \pi_{it}\).

Firm \(i\) chooses a decision rule that sets next period quantity \(\hat{q}_i\) as a function \(f_i\) of the current state \((q_i, q_{-i})\).

An essential aspect of a Markov perfect equilibrium is that each firm takes the decision rule of the other firm as known and given.

Given \(f_{-i}\), the Bellman equation of firm \(i\) is

\[
v_i(q_i, q_{-i}) = \max_{\hat{q}_i} \left\{ \pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i(\hat{q}_i, f_{-i}(q_{-i}, q_i)) \right\}
\]

(4.25)

**Definition** A Markov perfect equilibrium of the duopoly model is a pair of value functions \((v_1, v_2)\) and a pair of policy functions \((f_1, f_2)\) such that, for each \(i \in \{1, 2\}\) and each possible state,

- The value function \(v_i\) satisfies the Bellman equation (4.25)
- The maximizer on the right side of (4.25) is equal to \(f_i(q_i, q_{-i})\)

The adjective Markov denotes that the equilibrium decision rules depend only on the current values of the state variables, not other parts of their histories.

Perfect means complete, in the sense that the equilibrium is constructed by backward induction and hence builds in optimizing behavior for each firm at all possible future states.
These include many states that will not be reached when we iterate forward on the pair of equilibrium strategies \( f_i \) starting from a given initial state.

### Computation

One strategy for computing a Markov perfect equilibrium is iterating to convergence on pairs of Bellman equations and decision rules.

In particular, let \( v^j_i, f^j_i \) be the value function and policy function for firm \( i \) at the \( j \)-th iteration.

Imagine constructing the iterates

\[
v^{j+1}_i(q_i, q_{-i}) = \max_{q_i} \left\{ \sigma_i(q_i, q_{-i}, \hat{q}_i) + \beta v^j_i(\hat{q}_i, f_{-i}(q_{-i}, q_i)) \right\}
\]  

(4.26)

These iterations can be challenging to implement computationally.

However, they simplify for the case in which the one-period payoff functions are quadratic and the transition laws are linear, which takes us to our next topic.

#### 4.4.3 Linear Markov perfect equilibria

As we saw in the duopoly example, the study of Markov perfect equilibria in games with two players leads us to an interrelated pair of Bellman equations.

In linear quadratic dynamic games, these stacked Bellman equations become stacked Riccati equations with a tractable mathematical structure.

We will lay out that structure in a general setup and then apply it to some simple problems.

### Coupled linear regulator problems

We consider a general linear quadratic regulator game with two players.

For convenience, we start with a finite horizon formulation, where \( t_0 \) is the initial date and \( t_1 \) is the common terminal date.

Player \( i \) takes \( \{u_{-it}\} \) as given and minimizes

\[
s\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{ x_t'R_i x_t + u'_{it}Q_i u_{it} + u'_{-it}S_i u_{-it} + 2x_t'R_{it} u_{it} + 2u'_{-it}M_{it} u_{it} \right\}
\]

(4.27)

while the state evolves according to

\[
x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t}
\]

(4.28)

Here...
• $x_t$ is an $n \times 1$ state vector and $u_{it}$ is a $k_i \times 1$ vector of controls for player $i$
• $R_i$ is $n \times n$
• $S_i$ is $k_{-i} \times k_{-i}$
• $Q_i$ is $k_i \times k_i$
• $W_i$ is $n \times k_i$
• $M_i$ is $k_{-i} \times k_i$
• $A$ is $n \times n$
• $B_i$ is $n \times k_i$

**Computing Equilibrium**

We formulate a linear Markov perfect equilibrium as follows

Player $i$ employs linear decision rules $u_{it} = -F_{it}x_t$, where $F_{it}$ is a $k_i \times n$ matrix

A Markov perfect equilibrium is a pair of sequences $\{F_{1t}, F_{2t}\}$ over $t = t_0, \ldots, t_1 - 1$ such that

- $\{F_{1t}\}$ solves player 1s problem, taking $\{F_{2t}\}$ as given, and
- $\{F_{2t}\}$ solves player 2s problem, taking $\{F_{1t}\}$ as given

If we take $u_{2t} = -F_{2t}x_t$ and substitute it into (4.27) and (4.28), then player 1s problem becomes minimization of

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{ x_t' \Pi_{1t} x_t + u_{1t}' Q_1 u_{1t} + 2u_{1t}' \Gamma_{1t} x_t \right\}$$

subject to

$$x_{t+1} = A_{1t} x_t + B_{1t} u_{1t},$$

where

- $\Lambda_{it} := A - B_{-i} F_{-it}$
- $\Pi_{it} := R_i + F_{-it}' S_i F_{-it}$
- $\Gamma_{it} := W_i' - M_i' F_{-it}$

This is an LQ dynamic programming problem that can be solved by working backwards

The policy rule that solves this problem is

$$F_{1t} = (Q_1 + \beta B_{1t}' P_{1t+1} B_1)^{-1} (\beta B_{1t}' P_{1t+1} A_{1t} + \Gamma_{1t})$$

(4.31)
where $P_{1t}$ solves the matrix Riccati difference equation

$$
P_{1t} = \Pi_{1t} - (\beta B_1' P_{1t+1} A_{1t} + \Gamma_{1t})' (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} A_{1t} + \Gamma_{1t}) + \beta \Lambda_{1t}' P_{1t+1} A_{1t} \tag{4.32}
$$

Similarly, the policy that solves player 2's problem is

$$
F_{2t} = (Q_2 + \beta B_2' P_{2t+1} B_2)^{-1} (\beta B_2' P_{2t+1} A_{2t} + \Gamma_{2t}) \tag{4.33}
$$

where $P_{2t}$ solves

$$
P_{2t} = \Pi_{2t} - (\beta B_2' P_{2t+1} A_{2t} + \Gamma_{2t})' (Q_2 + \beta B_2' P_{2t+1} B_2)^{-1} (\beta B_2' P_{2t+1} A_{2t} + \Gamma_{2t}) + \beta \Lambda_{2t}' P_{2t+1} A_{2t} \tag{4.34}
$$

Here in all cases $t = t_0, \ldots, t_1 - 1$ and the terminal conditions are $P_{it_1} = 0$

The solution procedure is to use equations (4.31), (4.32), (4.33), and (4.34), and work backwards from time $t_1 - 1$

Since were working backwards, $P_{1t+1}$ and $P_{2t+1}$ are taken as given at each stage

Moreover, since

• some terms on the right hand side of (4.31) contain $F_{2t}$
• some terms on the right hand side of (4.33) contain $F_{1t}$

we need to solve these $k_1 + k_2$ equations simultaneously

**Key insight**

A key insight is that equations (4.31) and (4.33) are linear in $F_{1t}$ and $F_{2t}$

After these equations have been solved, we can take $F_{it}$ and solve for $P_{it}$ in (4.32) and (4.34)

**Infinite horizon**

We often want to compute the solutions of such games for infinite horizons, in the hope that the decision rules $F_{it}$ settle down to be time invariant as $t_1 \to +\infty$

In practice, we usually fix $t_1$ and compute the equilibrium of an infinite horizon game by driving $t_0 \to -\infty$

This is the approach we adopt in the next section

**Implementation**

We use the function `nash` from `QuantEcon.jl` that computes a Markov perfect equilibrium of the infinite horizon linear quadratic dynamic game in the manner described above
4.4.4 Application

Lets use these procedures to treat some applications, starting with the duopoly model

A duopoly model

To map the duopoly model into coupled linear-quadratic dynamic programming problems, define the state and controls as

\[
x_t := \begin{bmatrix} 1 \\ q_{1t} \\ q_{2t} \end{bmatrix} \quad \text{and} \quad u_{it} := q_{i,t+1} - q_{it}, \quad i = 1, 2
\]

If we write

\[
x_t^R_t x_t + u_{it}^Q u_{it}
\]

where \( Q_1 = Q_2 = \gamma \),

\[
R_1 := \begin{bmatrix} 0 & -\frac{a_0}{2} & 0 \\ -\frac{a_0}{2} & a_1 & \frac{a_1}{2} \\ 0 & \frac{a_1}{2} & 0 \end{bmatrix} \quad \text{and} \quad R_2 := \begin{bmatrix} 0 & 0 & -\frac{a_0}{2} \\ 0 & \frac{a_0}{2} & a_1 \\ -\frac{a_0}{2} & a_1 & 0 \end{bmatrix}
\]

then we recover the one-period payoffs in expression (4.24)

The law of motion for the state \( x_t \) is \( x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t} \) where

\[
A := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B_1 := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad B_2 := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}
\]

The optimal decision rule of firm \( i \) will take the form \( u_{it} = -F_i x_t \), inducing the following closed loop system for the evolution of \( x \) in the Markov perfect equilibrium:

\[
x_{t+1} = (A - B_1 F_1 - B_2 F_2) x_t \quad (4.35)
\]

Parameters and Solution

Consider the previously presented duopoly model with parameter values of:

- \( a_0 = 10 \)
- \( a_1 = 2 \)
- \( \beta = 0.96 \)
- \( \gamma = 12 \)

From these we compute the infinite horizon MPE using the preceding code.
Computed policies for firm 1 and firm 2:
F1 = [-0.668466 0.295125 0.0758467]
F2 = [-0.668466 0.0758467 0.295125]

One way to see that \( F_i \) is indeed optimal for firm \( i \) taking \( F_2 \) as given is to use QuantEcon.jl's \( LQ \) type.

In particular, let's take \( F_2 \) as computed above, plug it into (4.29) and (4.30) to get firm 1's problem and solve it using \( LQ \).

We hope that the resulting policy will agree with \( F_1 \) as computed above.
\[ A_1 = A - (B_2 + F_2) \]
\[ lq_1 = LQ(Q_1, R_1, A_1, B_1, \beta) \]
\[ P_{1_{ih}}, F_{1_{ih}}, d = \text{stationary_values}(lq_1) \]

\[ F_{1_{ih}} \]
\[ 1 \times 3 \text{ Array}\{\text{Float64}, 2\} : \]
\[ \begin{array}{ccc}
         -0.668466 & 0.295125 & 0.0758467 \\
    \end{array} \]

This is close enough for rock and roll, as they say in the trade

Indeed, \texttt{isapprox} agrees with our assessment

\[ \texttt{isapprox}(F_1, F_{1_{ih}}, \text{atol=1e-8, rtol=1e-8}) \]

\[ \text{true} \]

**Dynamics**

Let's now investigate the dynamics of price and output in this simple duopoly model under the MPE policies

Given our optimal policies \( F_1 \) and \( F_2 \), the state evolves according to (4.35)

The following program

- imports \( F_1 \) and \( F_2 \) from the previous program along with all parameters
- computes the evolution of \( x_t \) using (4.35)
- extracts and plots industry output \( q_t = q_{1t} + q_{2t} \) and price \( p_t = a_0 - a_1 q_t \)

```julia
using PyPlot
AF = A - B_1 * F_1 - B_2 * F_2
n = 20
x = Array\{Float64\}(3, n)
x[:, 1] = [1 1 1]
for t in 1:n-1
    x[:, t+1] = AF * x[:, t]
end
q1 = x[2, :]
q2 = x[3, :]
q = q1 + q2  # Total output, MPE
p = a_0 - a_1 * q  # Price, MPE

fig, ax = subplots(figsize=(9, 5.8))
ax[plot](q, "b-", lw=2, alpha=0.75, label="total output")
ax[plot](p, "g-", lw=2, alpha=0.75, label="price")
ax[:set_title]("Output and prices, duopoly MPE")
ax[legend](frameon=false)
```

Note that the initial condition has been set to \( q_{10} = q_{20} = 1.0 \)

The resulting figure looks as follows
To gain some perspective we can compare this to what happens in the monopoly case.

The first panel in the next figure compares output of the monopolist and industry output under the MPE, as a function of time.

The second panel shows analogous curves for price.
Here parameters are the same as above for both the MPE and monopoly solutions.

The monopolist initial condition is $q_0 = 2.0$ to mimic the industry initial condition $q_{10} = q_{20} = 1.0$ in the MPE case.

As expected, output is higher and prices are lower under duopoly than monopoly.
4.4.5 Exercises

Exercise 1

Replicate the pair of figures showing the comparison of output and prices for the monopolist and duopoly under MPE.

Parameters are as in duopoly_mpe.jl and you can use that code to compute MPE policies under duopoly.

The optimal policy in the monopolist case can be computed using QuantEcon.jl’s LQ type.

Exercise 2

In this exercise we consider a slightly more sophisticated duopoly problem.

It takes the form of infinite horizon linear quadratic game proposed by Judd [Jud90].

Two firms set prices and quantities of two goods interrelated through their demand curves.

Relevant variables are defined as follows:

- \( I_{it} \) = inventories of firm \( i \) at beginning of \( t \)
- \( q_{it} \) = production of firm \( i \) during period \( t \)
- \( p_{it} \) = price charged by firm \( i \) during period \( t \)
- \( S_{it} \) = sales made by firm \( i \) during period \( t \)
- \( E_{it} \) = costs of production of firm \( i \) during period \( t \)
- \( C_{it} \) = costs of carrying inventories for firm \( i \) during \( t \)

The firms cost functions are

- \( C_{it} = c_{i1} + c_{i2}I_{it} + 0.5c_{i3}I_{it}^2 \)
- \( E_{it} = e_{i1} + e_{i2}q_{it} + 0.5e_{i3}q_{it}^2 \) where \( e_{ij}, c_{ij} \) are positive scalars.

Inventories obey the laws of motion

\[
I_{i,t+1} = (1 - \delta)I_{it} + q_{it} - S_{it}
\]

Demand is governed by the linear schedule

\[
S_t = Dp_{it} + b
\]

where

- \( S_t = [S_{1t} \ S_{2t}]' \)
- \( D \) is a \( 2 \times 2 \) negative definite matrix and
- \( b \) is a vector of constants.
Firm $i$ maximizes the undiscounted sum

$$\lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T} (p_{it} s_{it} - E_{it} - C_{it})$$

We can convert this to a linear quadratic problem by taking

$$u_{it} = \begin{bmatrix} p_{it} \\ q_{it} \end{bmatrix} \quad \text{and} \quad x_t = \begin{bmatrix} I_{1t} \\ I_{2t} \\ 1 \end{bmatrix}$$

Decision rules for price and quantity take the form $u_{it} = -F_i x_t$

The Markov perfect equilibrium of Judds model can be computed by filling in the matrices appropriately.

The exercise is to calculate these matrices and compute the following figures.

The first figure shows the dynamics of inventories for each firm when the parameters are

\[
\delta = 0.02 \\
D = \begin{bmatrix} -1 & 0.5 \\ 0.5 & -1 \end{bmatrix} \\
b = [25, \ 25] \\
c1 = c2 = [1, -2, 1] \\
e1 = e2 = [10, 10, 3]
\]

Inventories trend to a common steady state.

If we increase the depreciation rate to $\delta = 0.05$, then we expect steady state inventories to fall.

This is indeed the case, as the next figure shows.
4.4.6 Solutions

Exercise 1

First lets compute the duopoly MPE under the stated parameters

```julia
# == Parameters == #
a0 = 10.0
a1 = 2.0
β = 0.96
γ = 12.0

# == In LQ form == #
A = eye(3)
B1 = [0.0, 1.0, 0.0]
B2 = [0.0, 0.0, 1.0]
R1 = [ 0.0 -a0 / 2.0 0.0;
      -a0 / 2.0 a1 a1 / 2.0;
      0.0 a1 / 2.0 0.0]
R2 = [ 0.0 0.0 -a0 / 2.0;
      0.0 0.0 a1 / 2.0;
      -a0 / 2.0 a1 / 2.0 a1]
Q1 = Q2 = γ
S1 = S2 = W1 = W2 = M1 = M2 = 0.0

# == Solve using QE's nnash function == #
P1, P2, P1, P2 = nnash(A, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2, beta=β)
```
Now we evaluate the time path of industry output and prices given initial condition $q_{10} = q_{20} = 1$

```
n = 20
x = Array{Float64}(3, n)
x[:, 1] = [1 1 1]
for t in 1:(n-1)
    x[:, t+1] = AF * x[:, t]
end
q1 = x[2, :]
q2 = x[3, :]
q = q1 + q2  # Total output, MPE
p = a0 - a1 * q  # Price, MPE
```

Next let's have a look at the monopoly solution.

For the state and control we take

$$x_t = q_t - \bar{q} \quad \text{and} \quad u_t = q_{t+1} - q_t$$

To convert to an LQ problem we set

$$R = a_1 \quad \text{and} \quad Q = \gamma$$

in the payoff function $x_t'Rx_t + u_t'Qu_t$ and

$$A = B = 1$$

in the law of motion $x_{t+1} = Ax_t + Bu_t$.

We solve for the optimal policy $u_t = -Fx_t$ and track the resulting dynamics of $\{q_t\}$, starting at $q_0 = 2.0$

```
R = a1
Q = \gamma
A = B = 1
lq_alt = LQ(Q, R, A, B, bet=\beta)
F, F1, d = stationary_values(lq_alt)
q_bar = a0 / (2.0 * a1)
qm = Array{Float64}(n)
qm[1] = 2
x0 = qm[1] - q_bar
x = x0
for i in 2:n
    x = A * x - B * F[1] * x
    qm[i] = float(x) + q_bar
end
pm = a0 - a1 * qm
```

Let's have a look at the different time paths.

```
fig, axes = subplots(2, 1, figsize=(9, 9))
ax = axes[1]
```
4.4. Markov Perfect Equilibrium
Exercise 2

We treat the case $\delta = 0.02$

\[
\delta = 0.02 \\
D = \begin{bmatrix} -1 & 0.5 \\ 0.5 & -1 \end{bmatrix} \\
b = [25, 25] \\
c1 = c2 = [1, -2, 1] \\
e1 = e2 = [10, 10, 3] \\
\delta_1 = 1-\delta
\]

Recalling that the control and state are

\[
\begin{bmatrix} p_{it} \\ q_{it} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} I_{1t} \\ I_{2t} \\ 1 \end{bmatrix}
\]

we set up the matrices as follows:

```julia
# == Create matrices needed to compute the Nash feedback equilibrium == #
A = [delta_1 0 -delta_1 * b[1]; 
     0 delta_1 -delta_1 * b[2]; 
     0 0 1]

B1 = delta_1 * [1 -D[1, 1]; 
                 0 -D[2, 1]; 
                 0 0]
B2 = delta_1 * [0 -D[1, 2]; 
                 1 -D[2, 2]; 
                 0 0]

R1 = [0.5 * c1[3] 0 0.5 * c1[2]; 
      0 0 0; 
      0.5 * c1[2] 0 c1[1]]
R2 = [0 0 0; 
      0 0.5 * c2[3] 0.5*c2[2]; 
      0 0.5 * c2[2] c2[1]]
Q1 = [-0.5*e1[3] 0; 
      0 D[1, 1]]
Q2 = [-0.5*e2[3] 0; 
      0 D[2, 2]]

S1 = zeros(2, 2) 
S2 = copy(S1)

W1 = [0.0 0.0; 
      0.0 0.0; 
      -0.5 * e1[2] b[1] / 2.0]
```

662 Chapter 4. Multiple Agent Models
W2 = [ 0.0  0.0;  
         0.0  0.0; 
        -0.5 * e2[2] b[2] / 2.0] 
M1 = [0.0  0.0;  
         0.0  D[1, 2] / 2.0] 
M2 = copy(M1) 
We can now compute the equilibrium using \texttt{qe.nnash}:

\begin{verbatim}
F1, F2, P1, P2 = nnash(A, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2)
println("\nFirm 1's feedback rule:\n")
println(F1)
println("\nFirm 2's feedback rule:\n")
println(F2)
\end{verbatim}

Firm 1's feedback rule:

\[
[0.243667 0.0272361 -6.82788; 0.392371 0.139696 -37.7341]\]

Firm 2's feedback rule:

\[
[0.0272361 0.243667 -6.82788; 0.139696 0.392371 -37.7341]\]

Now lets look at the dynamics of inventories, and reproduce the graph corresponding to $\delta = 0.02$:

\begin{verbatim}
n = 25
x = Array{Float64}(3, n)
x[:, 1] = [2 0 1]
for t in 1:(n-1)
    x[:, t+1] = AF * x[:, t]
end
I1 = x[1, :]
I2 = x[2, :]
fig, ax = subplots(figsize=(9, 5))
ax[plot](I1, "b-", lw=2, alpha=0.75, label="inventories, firm 1")
ax[plot](I2, "g-", lw=2, alpha=0.75, label="inventories, firm 2")
ax[set_title](latexstring("\delta = \$\delta\$"))
ax[legend]()
\end{verbatim}
4.5 Asset Pricing I: Finite State Models

Contents

• Asset Pricing I: Finite State Models
  – Overview
  – Pricing Models
  – Prices in the Risk Neutral Case
  – Asset Prices under Risk Aversion
  – Exercises
  – Solutions

A little knowledge of geometric series goes a long way – Robert E. Lucas, Jr.

Asset pricing is all about covariances – Lars Peter Hansen

4.5.1 Overview

An asset is a claim on one or more future payoffs
The spot price of an asset depends primarily on:
- the anticipated dynamics for the stream of income accruing to the owners
- attitudes to risk
- rates of time preference

In this lecture we consider some standard pricing models and dividend stream specifications.

We study how prices and dividend-price ratios respond in these different scenarios.

We also look at creating and pricing derivative assets by repackaging income streams.

Key tools for the lecture are:
- formulas for predicting future values of functions of a Markov state
- a formula for predicting the discounted sum of future values of a Markov state

### 4.5.2 Pricing Models

In what follows let \( \{d_t\}_{t \geq 0} \) be a stream of dividends:
- A time-\( t \) **cum-dividend** asset is a claim to the stream \( d_t, d_{t+1}, \ldots \)
- A time-\( t \) **ex-dividend** asset is a claim to the stream \( d_{t+1}, d_{t+2}, \ldots \)

Let’s look at some equations that we expect to hold for prices of assets under ex-dividend contracts (we will consider cum-dividend pricing in the exercises).

**Risk Neutral Pricing**

Our first scenario is risk-neutral pricing.

Let \( \beta = 1/(1 + \rho) \) be an intertemporal discount factor, where \( \rho \) is the rate at which agents discount the future.

The basic risk-neutral asset pricing equation for pricing one unit of an ex-dividend asset is:

\[
p_t = \beta \mathbb{E}_t [d_{t+1} + p_{t+1}] \tag{4.36}
\]

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 \( t \).

**Pricing with Random Discount Factor**

What happens if for some reason traders discount payouts differently depending on the state of the world?

Michael Harrison and David Kreps [HK79] and Lars Peter Hansen and Scott Richard [HR87] showed that in quite general settings the price of an ex-dividend asset obeys
\[ p_t = \mathbb{E}_t [m_{t+1} (d_{t+1} + p_{t+1})] \] (4.37)

for some **stochastic discount factor** \( m_{t+1} \)

The fixed discount factor \( \beta \) in (4.36) has been replaced by the random variable \( m_{t+1} \)

The way anticipated future payoffs are evaluated can now depend on various random outcomes

One example of this idea is that assets that tend to have good payoffs in bad states of the world might be regarded as more valuable

This is because they pay well when the funds are more urgently needed

We give examples of how the stochastic discount factor has been modeled below

### Asset Pricing and Covariances

Recall that, from the definition of a conditional covariance \( \text{cov}_t(x_{t+1}, y_{t+1}) \), we have

\[ \mathbb{E}_t(x_{t+1}y_{t+1}) = \text{cov}_t(x_{t+1}, y_{t+1}) + \mathbb{E}_t x_{t+1}\mathbb{E}_t y_{t+1} \] (4.38)

If we apply this definition to the asset pricing equation (4.37) we obtain

\[ p_t = \mathbb{E}_t m_{t+1} \mathbb{E}_t (d_{t+1} + p_{t+1}) + \text{cov}_t (m_{t+1}, d_{t+1} + p_{t+1}) \] (4.39)

It is useful to regard equation (4.39) as a generalization of equation (4.36)

* In equation (4.36), the stochastic discount factor \( m_{t+1} = \beta \), a constant
* In equation (4.36), the covariance term \( \text{cov}_t (m_{t+1}, d_{t+1} + p_{t+1}) \) is zero because \( m_{t+1} = \beta \)

Equation (4.39) asserts that the covariance of the stochastic discount factor with the one period payout \( d_{t+1} + p_{t+1} \) is an important determinant of the price \( p_t \)

We give examples of some models of stochastic discount factors that have been proposed later in this lecture and also in a later lecture

### The Price-Dividend Ratio

Aside from prices, another quantity of interest is the **price-dividend ratio** \( v_t := p_t / d_t \)

Let’s write down an expression that this ratio should satisfy

We can divide both sides of (4.37) by \( d_t \) to get

\[ v_t = \mathbb{E}_t \left[ m_{t+1} \frac{d_{t+1}}{d_t} (1 + v_{t+1}) \right] \] (4.40)

Below we discuss the implication of this equation
4.5.3 Prices in the Risk Neutral Case

What can we say about price dynamics on the basis of the models described above?
The answer to this question depends on

1. the process we specify for dividends
2. the stochastic discount factor and how it correlates with dividends

For now let’s focus on the risk neutral case, where the stochastic discount factor is constant, and study how prices depend on the dividend process

Example 1: Constant dividends

The simplest case is risk neutral pricing in the face of a constant, non-random dividend stream $d_t = d > 0$

Removing the expectation from (4.36) and iterating forward gives

\[
p_t = \beta(d + p_{t+1}) \\
= \beta(d + \beta(d + p_{t+2})) \\
\vdots \\
= \beta(d + \beta d + \beta^2 d + \cdots + \beta^{k-2} d + \beta^{k-1} p_{t+k})
\]

Unless prices explode in the future, this sequence converges to

\[
\bar{p} := \frac{\beta d}{1 - \beta}
\]  
(4.41)

This price is the equilibrium price in the constant dividend case

Indeed, simple algebra shows that setting $p_t = \bar{p}$ for all $t$ satisfies the equilibrium condition $p_t = \beta(d + p_{t+1})$

Example 2: Dividends with deterministic growth paths

Consider a growing, non-random dividend process $d_{t+1} = g d_t$ where $0 < g \beta < 1$

While prices are not usually constant when dividends grow over time, the price dividend-ratio might be

If we guess this, substituting $v_t = v$ into (4.40) as well as our other assumptions, we get $v = \beta g (1 + v)$

Since $\beta g < 1$, we have a unique positive solution:

\[
v = \frac{\beta g}{1 - \beta g}
\]

The price is then

\[
p_t = \frac{\beta g}{1 - \beta g} d_t
\]
If, in this example, we take \( g = 1 + \kappa \) and let \( \rho := 1/\beta - 1 \), then the price becomes

\[
p_t = \frac{1 + \kappa}{\rho - \kappa} d_t
\]

This is called the \textit{Gordon formula}.

\textbf{Example 3: Markov growth, risk neutral pricing}

Next we consider a dividend process

\[
d_{t+1} = g_{t+1} d_t
\]

The stochastic growth factor \( \{g_t\} \) is given by

\[
g_t = g(X_t), \quad t = 1, 2, \ldots
\]

where

1. \( \{X_t\} \) is a finite Markov chain with state space \( S \) and transition probabilities

\[
P(x, y) := \mathbb{P}\{X_{t+1} = y \mid X_t = x\} \quad (x, y \in S)
\]

2. \( g \) is a given function on \( S \) taking positive values

You can think of

- \( S \) as \( n \) possible states of the world and \( X_t \) as the current state
- \( g \) as a function that maps a given state \( X_t \) into a growth factor \( g_t = g(X_t) \) for the endowment
- \( \ln g_t = \ln(d_{t+1}/d_t) \) is the growth rate of dividends

(For a refresher on notation and theory for finite Markov chains see \textit{this lecture})

The next figure shows a simulation, where

- \( \{X_t\} \) evolves as a discretized AR1 process produced using \textit{Tauchens method}
- \( g_t = \exp(X_t) \), so that \( \ln g_t = X_t \) is the growth rate

```julia
using QuantEcon
using Plots
using LaTeXStrings
pyplot()

n = 25
mc = tauchen(n, 0.96, 0.25)
sim_length = 80

x_series = simulate(mc, sim_length; init=round(Int, n / 2))
g_series = exp.(x_series)
d_series = cumprod(g_series) # assumes \( d_0 = 1 \)
```
Pricing

To obtain asset prices in this setting, let's adapt our analysis from the case of deterministic growth. In that case, we found that $v$ is constant. This encourages us to guess that, in the current case, $v_t$ is constant given the state $X_t$.

In other words, we are looking for a fixed function $v$ such that the price-dividend ratio satisfies $v_t = v(X_t)$.

We can substitute this guess into (4.40) to get:

$$v(X_t) = \beta \mathbb{E}_t\left[g(X_{t+1})(1 + v(X_{t+1}))\right]$$

If we condition on $X_t = x$, this becomes:

$$v(x) = \beta \sum_{y \in S} g(y)(1 + v(y))P(x, y)$$

or

4.5. Asset Pricing I: Finite State Models
\[ v(x) = \beta \sum_{y \in S} K(x, y)(1 + v(y)) \quad \text{where} \quad K(x, y) := g(y)P(x, y) \]

(4.43)

Suppose that there are \( n \) possible states \( x_1, \ldots, x_n \).

We can then think of (4.43) as \( n \) stacked equations, one for each state, and write it in matrix form as

\[ v = \beta K (\mathbf{1} + v) \]

(4.44)

Here

- \( v \) is understood to be the column vector \((v(x_1), \ldots, v(x_n))'\)
- \( K \) is the matrix \((K(x_i, x_j))_{1 \leq i, j \leq n}\)
- \( \mathbf{1} \) is a column vector of ones

When does (4.44) have a unique solution?

From the Neumann series lemma and Gelfands formula, this will be the case if \( \beta K \) has spectral radius strictly less than one.

In other words, we require that the eigenvalues of \( K \) be strictly less than \( \beta^{-1} \) in modulus.

The solution is then

\[ v = (I - \beta K)^{-1} \beta K \mathbf{1} \]

(4.45)

**Code**

Lets calculate and plot the price-dividend ratio at a set of parameters.

As before, well generate \( \{X_t\} \) as a discretized AR1 process and set \( g_t = \exp(X_t) \).

Heres the code, including a test of the spectral radius condition.

```julia
n = 25  # size of state space
β = 0.9
mc = tauchen(n, 0.96, 0.02)
K = mc.p .* exp.(mc.state_values)'
I = eye(n)
v = (I - β * K) \ (β * K + ones(n, 1))

plot(mc.state_values,
     v,
     lw=2,
     ylabel="price-dividend ratio",
xlabel="state",
)
```

Chapter 4. Multiple Agent Models
Why does the price-dividend ratio increase with the state?

The reason is that this Markov process is positively correlated, so high current states suggest high future states.

Moreover, dividend growth is increasing in the state.

Anticipation of high future dividend growth leads to a high price-dividend ratio.

### 4.5.4 Asset Prices under Risk Aversion

Now let's turn to the case where agents are risk averse.

We'll price several distinct assets, including:

- The price of an endowment stream
- A consol (a type of bond issued by the UK government in the 19th century)
- Call options on a consol

\[
\alpha = 0.7, \quad \text{label} = \text{L"$v"$)}
\]
Pricing a Lucas tree

Let's start with a version of the celebrated asset pricing model of Robert E. Lucas, Jr. [Luc78].

As in [Luc78], suppose that the stochastic discount factor takes the form

\[ m_{t+1} = \beta \frac{u'(c_{t+1})}{u'(c_t)} \]  

(4.46)

where \( u \) is a concave utility function and \( c_t \) is time \( t \) consumption of a representative consumer.

(A derivation of this expression is given in a later lecture.)

Assume the existence of an endowment that follows (4.42)

The asset being priced is a claim on the endowment process.

Following [Luc78], suppose further that in equilibrium, consumption is equal to the endowment, so that \( d_t = c_t \) for all \( t \).

For utility, we will assume the constant relative risk aversion (CRRA) specification

\[ u(c) = \frac{c^{1-\gamma}}{1-\gamma} \text{ with } \gamma > 0 \]  

(4.47)

When \( \gamma = 1 \) we let \( u(c) = \ln c \).

Inserting the CRRA specification into (4.46) and using \( c_t = d_t \) gives

\[ m_{t+1} = \beta \left( \frac{c_{t+1}}{c_t} \right)^{-\gamma} = \beta g_{t+1} \]  

(4.48)

Substituting this into (4.40) gives the price-dividend ratio formula

\[ v(X_t) = \beta \mathbb{E}_t \left[ g(X_{t+1})^{1-\gamma} (1 + v(X_{t+1})) \right] \]

Conditioning on \( X_t = x \), we can write this as

\[ v(x) = \beta \sum_{y \in S} g(y)^{1-\gamma} (1 + v(y)) P(x, y) \]

If we let

\[ J(x, y) := g(y)^{1-\gamma} P(x, y) \]

then we can rewrite in vector form as

\[ v = \beta J \nu + v \]

Assuming that the spectral radius of \( J \) is strictly less than \( \beta^{-1} \), this equation has the unique solution

\[ v = (I - \beta J)^{-1} \beta \nu \]  

(4.49)

We will define a function \texttt{tree_price} to solve for \( v \) given parameters stored in the type \texttt{AssetPriceModel}.
# @authors: Spencer Lyon, Tom Sargent, John Stachurski

# A default Markov chain for the state process
\( \rho = 0.9 \)
\( \sigma = 0.02 \)
n = 25
default_mc = tauchen(n, \rho, \sigma)

mutable struct AssetPriceModel{TF<:AbstractFloat, TI<:Integer}
    \( \beta \) :: TF  # Discount factor
    \( \gamma \) :: TF  # Coefficient of risk aversion
    mc :: MarkovChain  # State process
    n :: TI  # Number of states
    g :: Function  # Function mapping states into growth rates
end

function AssetPriceModel(;\( \beta \)::AbstractFloat=0.96,
    \( \gamma \)::AbstractFloat=2.0,
    mc::MarkovChain=default_mc,
    g::Function=exp)
    n = size(mc.p)[1]
    return AssetPriceModel(\( \beta \), \( \gamma \), mc, n, g)
end

"""
Stability test for a given matrix \( Q \).
"""
function test_stability(ap::AssetPriceModel, Q::Matrix)
    sr = maximum(abs, eigvals(Q))
    if sr >= 1 / ap.\( \beta \)
        msg = "Spectral radius condition failed with radius = \$sr"
        throw(ArgumentError(msg))
    end
end

"""
Computes the price-dividend ratio of the Lucas tree.
"""
function tree_price(ap::AssetPriceModel)
    \( \beta \), \( \gamma \), P, y = ap.\( \beta \), ap.\( \gamma \), ap.mc.p, ap.mc.state_values
    y = reshape(y, 1, ap.n)
    J = P .\* ap.g.(y).\(^{(\gamma - 1)}\)

    # Make sure that a unique solution exists
end
test_stability(ap, J)

# == Compute v == #
I = eye(ap.n)
Ones = ones(ap.n)
v = (I - β * J) \ (β * J + Ones)

return v
end

Here’s a plot of $v$ as a function of the state for several values of $\gamma$, with a positively correlated Markov process and $g(x) = \exp(x)$

$\gamma$s = [1.2, 1.4, 1.6, 1.8, 2.0]
ap = AssetPriceModel()
states = ap.mc.state_values

lines = []
labels = []

for $\gamma$ in $\gamma$s
    ap.γ = $\gamma$
    v = tree_price(ap)
    label="$\gamma = $γ"
    push!(labels, label)
    push!(lines, v)
end

plot(lines,
     labels=reshape(labels, 1, length(labels)),
     title="Price-dividend ratio as a function of the state",
     ylabel="price-dividend ratio",
     xlabel="state")
Notice that $v$ is decreasing in each case.

This is because, with a positively correlated state process, higher states suggest higher future consumption growth.

In the stochastic discount factor (4.48), higher growth decreases the discount factor, lowering the weight placed on future returns.

**Special cases**

In the special case $\gamma = 1$, we have $J = P$.

Recalling that $P^i i = i$ for all $i$ and applying Neumanns geometric series lemma, we are led to

$$v = \beta (I - \beta P)^{-1} i = \beta \sum_{i=0}^{\infty} \beta^i P^i i = \beta \frac{1}{1 - \beta} i$$

Thus, with log preferences, the price-dividend ratio for a Lucas tree is constant.

Alternatively, if $\gamma = 0$, then $J = K$ and we recover the risk neutral solution (4.45).

This is as expected, since $\gamma = 0$ implies $u(c) = c$ (and hence agents are risk neutral).
A Risk-Free Consol

Consider the same pure exchange representative agent economy

A risk-free consol promises to pay a constant amount $\zeta > 0$ each period.

Recycling notation, let $p_t$ now be the price of an ex-coupon claim to the consol.

An ex-coupon claim to the consol entitles the owner at the end of period $t$ to

- $\zeta$ in period $t+1$, plus
- the right to sell the claim for $p_{t+1}$ next period.

The price satisfies (4.37) with $d_t = \zeta$, or

$$p_t = \mathbb{E}_t \left[ m_{t+1}(\zeta + p_{t+1}) \right]$$

We maintain the stochastic discount factor (4.48), so this becomes

$$p_t = \mathbb{E}_t \left[ \beta g_{t+1}^{-\gamma}(\zeta + p_{t+1}) \right]$$

(4.50)

Guessing a solution of the form $p_t = p(X_t)$ and conditioning on $X_t = x$, we get

$$p(x) = \beta \sum_{y \in S} g(y)^{-\gamma}(\zeta + p(y))P(x, y)$$

Letting $M(x, y) = P(x, y)g(y)^{-\gamma}$ and rewriting in vector notation yields the solution

$$p = (I - \beta M)^{-1}\beta M\zeta\kappa$$

(4.51)

The above is implemented in the function `consol_price`

```julia
function consol_price(ap::AssetPriceModel, ζ::AbstractFloat)
    # == Simplify names, set up matrices == #
    β, γ, P, y = ap.β, ap.γ, ap.mc.p, ap.mc.state_values
    y = reshape(y, 1, ap.n)
    M = P .* ap.g.(y).^(-γ)

    # == Make sure that a unique solution exists == #
    test_stability(ap, M)

    # == Compute price == #
    I = eye(ap.n)
    Ones = ones(ap.n)
    p = (I - β * M) \ (β * ζ * M * Ones)

    return p
end
```

Chapter 4. Multiple Agent Models
Pricing an Option to Purchase the Consol

Let's now price options of varying maturity that give the right to purchase a consol at a price $p_S$

An infinite horizon call option

We want to price an infinite horizon option to purchase a consol at a price $p_S$

The option entitles the owner at the beginning of a period either to

1. purchase the bond at price $p_S$ now, or
2. Not to exercise the option now but to retain the right to exercise it later

Thus, the owner either exercises the option now, or chooses not to exercise and wait until next period

This is termed an infinite-horizon call option with strike price $p_S$

The owner of the option is entitled to purchase the consol at the price $p_S$ at the beginning of any period, after the coupon has been paid to the previous owner of the bond

The fundamentals of the economy are identical with the one above, including the stochastic discount factor and the process for consumption

Let $w(X_t, p_S)$ be the value of the option when the time $t$ growth state is known to be $X_t$ but before the owner has decided whether or not to exercise the option at time $t$ (i.e., today)

Recalling that $p(X_t)$ is the value of the consol when the initial growth state is $X_t$, the value of the option satisfies

$$w(X_t, p_S) = \max \left\{ \beta \mathbb{E}_t \frac{u'(c_{t+1})}{u'(c_t)} w(X_{t+1}, p_S), p(X_t) - p_S \right\}$$

The first term on the right is the value of waiting, while the second is the value of exercising now

We can also write this as

$$w(x, p_S) = \max \left\{ \beta \sum_{y \in S} P(x, y) g(y)^{-\gamma} w(y, p_S), p(x) - p_S \right\} \quad (4.52)$$

With $M(x, y) = P(x, y) g(y)^{-\gamma}$ and $w$ as the vector of values $(w(x_i), p_S)_{i=1}^n$, we can express (4.52) as the nonlinear vector equation

$$w = \max \{ \beta M w, p - p_S \} \quad (4.53)$$

To solve (4.53), form the operator $T$ mapping vector $w$ into vector $Tw$ via

$$Tw = \max \{ \beta M w, p - p_S \}$$

Start at some initial $w$ and iterate to convergence with $T$

We can find the solution with the following function call_option

4.5. Asset Pricing I: Finite State Models
### Function `call_option` Definition

The function `call_option` computes the price of a perpetual call option on a consol bond.

```julia
function call_option(ap::AssetPriceModel, ζ::AbstractFloat, p_ →s::AbstractFloat, =1e-7)
    # == Simplify names, set up matrices == #
    β, γ, P, y = ap.β, ap.γ, ap.mc.p, ap.mc.state_values
    y = reshape(y, 1, ap.n)
    M = P .* ap.g.(y).^(-γ)

    # == Make sure that a unique console price exists == #
    test_stability(ap, M)

    # == Compute option price == #
    p = consol_price(ap, ζ)
    w = zeros(ap.n, 1)
    error = + 1
    while (error > )
        # == Maximize across columns == #
        w_new = max.(β * M * w, p - p_s)
        # == Find maximal difference of each component and update == #
        error = maximum(abs, w - w_new)
        w = w_new
    end

    return w
end
```

### Example Usage

Here's a plot of $w$ compared to the consol price when $P_S = 40$

```julia
ap = AssetPriceModel(β=0.9)
ζ = 1.0
strike_price = 40.0

x = ap.mc.state_values
p = consol_price(ap, ζ)
w = call_option(ap, ζ, strike_price)

plot(x, p, color="blue", lw=2, xlabel="state", label="consol price")
plot!(x, w, color="green", lw=2, label="value of call option")
```

---

Chapter 4. Multiple Agent Models
In large states the value of the option is close to zero

This is despite the fact the Markov chain is irreducible and low states where the consol prices is high will eventually be visited

The reason is that $\beta = 0.9$, so the future is discounted relatively rapidly

**Risk Free Rates**

Let's look at risk free interest rates over different periods

**The one-period risk-free interest rate**

As before, the stochastic discount factor is $m_{t+1} = \beta g_{t+1}^\gamma$.

It follows that the reciprocal $R_t^{-1}$ of the gross risk-free interest rate $R_t$ in state $x$ is

$$\mathbb{E}_t m_{t+1} = \beta \sum_{y \in S} P(x, y) g(y)^{-\gamma}$$

We can write this as

$$m_1 = \beta M \kappa$$

where the $i$-th element of $m_1$ is the reciprocal of the one-period gross risk-free interest rate in state $x_i$
Other terms

Let \( m_j \) be an \( n \times 1 \) vector whose \( i \) th component is the reciprocal of the \( j \)-period gross risk-free interest rate in state \( x_i \)

Then \( m_1 = \beta M \), and \( m_{j+1} = M m_j \) for \( j \geq 1 \)

4.5.5 Exercises

Exercise 1

In the lecture, we considered **ex-dividend assets**

A **cum-dividend** asset is a claim to the stream \( d_t, d_{t+1}, \ldots \)

Following (4.36), find the risk-neutral asset pricing equation for one unit of a cum-dividend asset

With a constant, non-random dividend stream \( d_t = d > 0 \), what is the equilibrium price of a cum-dividend asset?

With a growing, non-random dividend process \( d_t = g d_t \) where \( 0 < g \beta < 1 \), what is the equilibrium price of a cum-dividend asset?

Exercise 2

Consider the following primitives

```plaintext
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]
\gamma = 2.0
\beta = 0.94
\zeta = 1.0
```

Let \( g \) be defined by \( g(x) = x \) (that is, \( g \) is the identity map)

Compute the price of the Lucas tree

Do the same for

- the price of the risk-free consol when \( \zeta = 1 \)
- the call option on the consol when \( \zeta = 1 \) and \( p_S = 150.0 \)

Exercise 3

Let's consider finite horizon call options, which are more common than the infinite horizon variety

Finite horizon options obey functional equations closely related to (4.52)

A \( k \) period option expires after \( k \) periods
If we view today as date zero, a $k$ period option gives the owner the right to exercise the option to purchase the risk-free consol at the strike price $p_S$ at dates $0, 1, \ldots, k - 1$

The option expires at time $k$

Thus, for $k = 1, 2, \ldots$, let $w(x, k)$ be the value of a $k$-period option

It obeys

$$w(x, k) = \max \left\{ \beta \sum_{y \in S} P(x, y)g(y)^{-\gamma}w(y, k - 1), \ p(x) - p_S \right\}$$

where $w(x, 0) = 0$ for all $x$

We can express the preceding as the sequence of nonlinear vector equations

$$w_k = \max \{ \beta M w_{k-1}, p - p_S \} \quad k = 1, 2, \ldots \quad \text{with } w_0 = 0$$

Write a function that computes $w_k$ for any given $k$

Compute the value of the option with $k = 5$ and $k = 25$ using parameter values as in Exercise 1

Is one higher than the other? Can you give intuition?

### 4.5.6 Solutions

#### Exercise 1

```julia
n = 5
P = 0.0125 .* ones(n, n)
P = P .+ diagm(0.95 - 0.0125 .* ones(5))
s = [0.95, 0.975, 1.0, 1.025, 1.05]  # state values
mc = MarkovChain(P, s)
γ = 2.0
β = 0.94
ζ = 1.0
p_s = 150.0

150.0

Next we create an instance of `AssetPriceModel` to feed into the functions.

```julia
ap = AssetPriceModel(β=β, mc=mc, γ=γ, g=x -> x)

v = tree_price(ap)
println("Lucas Tree Prices: \$v\n")
```

Lucas Tree Prices: [29.474, 21.9357, 17.5714, 14.7252, 12.7222]
v_consol = consol_price(ap, 1.0)
println("Consol Bond Prices: \$(v_consol)\n")

Consol Bond Prices: [753.871, 242.551, 148.676, 109.251, 87.5686]

w = call_option(ap, ζ, p_s)

Exercise 2

Here is a suitable function:

```julia
function finite_horizon_call_option(ap::AssetPriceModel,
ζ::AbstractFloat,
p_s::AbstractFloat,
k::Int)
  # == Simplify names, set up matrices == #
  β, γ, P, y = ap.β, ap.γ, ap.mc.p, ap.mc.state_values
  y = y'
  M = P .* ap.g(y).^(-γ)
  # == Make sure that a unique console price exists == #
  test_stability(ap, M)
  # == Compute option price == #
  p = consol_price(ap, ζ)
  w = zeros(ap.n, 1)
  for i in 1:k
    # == Maximize across columns == #
    w = max.(β * M * w, p - p_s)
  end
  return w
end
```

```julia
lines = []
labels = []
for k in [5, 25]
  w = finite_horizon_call_option(ap, ζ, p_s, k)
  push!(lines, w)
  push!(labels, "k = \$(k)")
end
plot(lines, labels=reshape(labels, 1, length(labels)))
```
Not surprisingly, the option has greater value with larger $k$. This is because the owner has a longer time horizon over which he or she may exercise the option.

### 4.6 Asset Pricing II: The Lucas Asset Pricing Model

#### Contents

- Asset Pricing II: The Lucas Asset Pricing Model
  - Overview
  - The Lucas Model
  - Exercises
  - Solutions

#### 4.6.1 Overview

As stated in an earlier lecture, an asset is a claim on a stream of prospective payments.

What is the correct price to pay for such a claim?
The elegant asset pricing model of Lucas [Luc78] attempts to answer this question in an equilibrium setting with risk averse agents.

While we mentioned some consequences of Lucas model earlier, it is now time to work through the model more carefully, and try to understand where the fundamental asset pricing equation comes from.

A side benefit of studying Lucas model is that it provides a beautiful illustration of model building in general and equilibrium pricing in competitive models in particular.

Another difference to our first asset pricing lecture is that the state space and shock will be continuous rather than discrete.

### 4.6.2 The Lucas Model

Lucas studied a pure exchange economy with a representative consumer (or household), where

- *Pure exchange* means that all endowments are exogenous
- *Representative* consumer means that either
  - there is a single consumer (sometimes also referred to as a household), or
  - all consumers have identical endowments and preferences

Either way, the assumption of a representative agent means that prices adjust to eradicate desires to trade. This makes it very easy to compute competitive equilibrium prices.

#### Basic Setup

Let’s review the setup.

#### Assets

There is a single productive unit that costlessly generates a sequence of consumption goods \( \{y_t\}_{t=0}^{\infty} \). Another way to view \( \{y_t\}_{t=0}^{\infty} \) is as a *consumption endowment* for this economy.

We will assume that this endowment is Markovian, following the exogenous process

\[
y_{t+1} = G(y_t, \xi_{t+1})
\]

Here \( \{\xi_t\} \) is an iid shock sequence with known distribution \( \phi \) and \( y_t \geq 0 \).

An asset is a claim on all or part of this endowment stream.

The consumption goods \( \{y_t\}_{t=0}^{\infty} \) are nonstorable, so holding assets is the only way to transfer wealth into the future.

For the purposes of intuition, it’s common to think of the productive unit as a tree that produces fruit.

Based on this idea, a Lucas tree is a claim on the consumption endowment.
Consumers

A representative consumer ranks consumption streams \( \{c_t\} \) according to the time separable utility functional

\[
\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \quad (4.54)
\]

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?

Well 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 well impose equilibrium constraints and try to back out prices

In the consumer problem, the consumers control variable is the share \( \pi_t \) of the claim held in each period

Thus, the consumer problem is to maximize \((4.54)\) 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

\[
c + \pi' p(y) \leq \pi y + \pi p(y)
\]

We can invoke the fact that utility is increasing to claim equality in (4.55) and hence eliminate the constraint, obtaining

\[
v(\pi, y) = \max_{\pi'} \left\{ u[\pi(y + p(y)) - \pi' p(y)] + \beta \int v(\pi', G(y, z)) \phi(dz) \right\}
\]

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 (4.56) 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 (4.56) 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

$$p(y) = \beta \int \frac{u'[G(y, z)]}{u'(y)} [G(y, z) + p(G(y, z))]\phi(dz) \quad (4.57)$$

In sequential rather than functional notation, we can also write this as

$$p_t = E_t \left[ \beta \frac{u'(c_{t+1})}{u'(c_t)} (y_{t+1} + p_{t+1}) \right] \quad (4.58)$$

This is the famous consumption-based asset pricing equation

Before discussing it further we want to solve out for prices

Solving the Model

Equation (4.57) is a functional equation in the unknown function $p$

The solution is an equilibrium price function $p^*$

Lets look at how to obtain it
Setting up the problem

Instead of solving for it directly we’ll follow Lucas’s indirect approach, first setting

$$f(y) := u'(y)p(y)$$

so that (4.57) becomes

$$f(y) = h(y) + \beta \int f[G(y, z)] \phi(dz)$$

(4.60)

Here \(h(y) := \beta \int u'[G(y, z)]G(y, z)\phi(dz)\) is a function that depends only on the primitives.

Equation (4.60) is a functional equation in \(f\).

The plan is to solve out for \(f\) and convert back to \(p\) via (4.59).

To solve (4.60) well use a standard method: convert it to a fixed point problem.

First we introduce the operator \(T\) mapping \(f\) into \(Tf\) as defined by

$$(Tf)(y) = h(y) + \beta \int f[G(y, z)]\phi(dz)$$

(4.61)

The reason we do this is that a solution to (4.60) 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

$$\|Tf - Tg\| \leq \alpha \|f - g\|, \quad \forall f, g \in cb\mathbb{R}_+$$

(4.62)
Here \( \|h\| := \sup_{x \in \mathbb{R}^+} |h(x)| \)

To see that (4.62) is valid, pick any \( f, g \in \mathbb{C}b\mathbb{R}^+ \) and any \( y \in \mathbb{R}^+ \)

Observe that, since integrals get larger when absolute values are moved to the inside,

\[
|Tf(y) - Tg(y)| = \left| \beta \int f(G(y, z))\phi(dz) - \beta \int g(G(y, z))\phi(dz) \right|
\leq \beta \int |f(G(y, z)) - g(G(y, z))| \phi(dz)
\leq \beta \int \|f - g\| \phi(dz)
= \beta \|f - g\|
\]

Since the right hand side is an upper bound, taking the sup over all \( y \) on the left hand side gives (4.62) with \( \alpha := \beta \)

**Computation – An Example**

The preceding discussion tells that we can compute \( f^* \) by picking any arbitrary \( f \in \mathbb{C}b\mathbb{R}^+ \) 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`

We repeat it here for convenience

```julia
#=
@authors: Spencer Lyon <spencer.lyon@nyu.edu>, John Stachurski
=#
using QuantEcon
using Distributions

""
The Lucas asset pricing model --- parameters and grid data
""
struct LucasTree{TF<:AbstractFloat}
    \gamma::TF    # coefficient of risk aversion
    \beta::TF    # Discount factor in (0, 1)
    \alpha::TF   # Correlation coefficient in the shock process
    \sigma::TF   # Volatility of shock process
    \::Distribution    # Distribution for shock process
    grid::Vector{TF} # Grid of points on which to evaluate prices
    shocks::Vector{TF} # Draws of the shock
end
```

4.6. Asset Pricing II: The Lucas Asset Pricing Model
The \( h \) function represented as a vector

```
Vector{Tf}
```

## Constructor for the Lucas asset pricing model

```
function LucasTree(;\gamma::AbstractFloat=2.0,
  \beta::AbstractFloat=0.95,
  \alpha::AbstractFloat=0.9,
  \sigma::AbstractFloat=0.1,
  grid_size::Integer=100)

  = LogNormal(0.0, \sigma)
shocks = rand(, 500)

  # == build a grid with mass around stationary distribution == #
ssd = \sigma / sqrt(1 - \alpha^2)
grid_min, grid_max = exp(-4 * ssd), exp(4 * ssd)
grid = collect(linspace(grid_min, grid_max, grid_size))

  # == set \( h(y) = \beta + \int u'(G(y,z)) G(y,z) \, dz \) == #
h = similar(grid)
for (i, y) in enumerate(grid)
  h[i] = \beta + mean((y^\alpha .* shocks).^(1 - \gamma))
end

return LucasTree(\gamma, 
  \beta, 
  \alpha, 
  \sigma, 
  grid, 
  shocks, 
  h)
end
```

The approximate Lucas operator, which computes and returns updated function \( T_f \) on the grid points.

```
function lucas_operator(lt::LucasTree, f::Vector)

  # == unpack names == #
  grid, \alpha, \beta, h = lt.grid, lt.\alpha, lt.\beta, lt.h
z = lt.shocks

  Af = LinInterp(grid, f)

  Tf = [h[i] + \beta + mean(Af.(grid[i]^{\alpha}.*z)) for i in 1:length(grid)]
return Tf
end
```
Compute the equilibrium price function associated with Lucas tree `lt`

```julia
function solve_lucas_model(lt::LucasTree;
    tol::AbstractFloat=1e-6,
    max_iter::Integer=500)

    # == simplify notation == #
    grid, γ = lt.grid, lt.γ

    i = 0
    f = zeros(grid)  # Initial guess of f
    error = tol + 1

    while (error > tol) && (i < max_iter)
        f_new = lucas_operator(lt, f)
        error = maximum(abs, f_new - f)
        f = f_new
        i += 1
    end

    # p(y) = f(y) * y ^ γ
    price = f .* grid.ˆγ

    return price
end
```

An example of usage is given in the docstring and repeated here:

```julia
tree = LucasTree(γ=2.0, β=0.95, α=0.90, σ=0.1)
price_vals = solve_lucas_model(tree);
```

Here's the resulting price function:

```julia
using PyPlot

plt[:plot](figsize=(12, 8))
plt[:plot](tree.grid, price_vals, label=L"p\ast(y)")
plt[:xlabel](L"y")
plt[:ylabel]("price")
plt[:legend]()
```

4.6. Asset Pricing II: The Lucas Asset Pricing Model
The price is increasing, even if we remove all serial correlation from the endowment process

The reason is that a larger current endowment reduces current marginal utility

The price must therefore rise to induce the household to consume the entire endowment (and hence satisfy the resource constraint)

What happens with a more patient consumer?

Here the orange line corresponds to the previous parameters and the green line is price when $\beta = 0.98$
We see that when consumers are more patient the asset becomes more valuable, and the price of the Lucas tree shifts up.

Exercise 1 asks you to replicate this figure.

### 4.6.3 Exercises

**Exercise 1**

Replicate the figure to show how discount rates affect prices.

### 4.6.4 Solutions

```julia
for β in (.95, 0.98)
    tree = LucasTree(;β=β)
    grid = tree.grid
    price_vals = solve_lucas_model(tree)
    plt[:plot](grid, price_vals, label=latexstring("\$ \beta = $β\$"))
end
plt[:legend]()
```
4.7 Asset Pricing III: Incomplete Markets

This lecture describes a version of a model of Harrison and Kreps [HK78]
The model determines the price of a dividend-yielding asset that is traded by two types of self-interested investors

The model features

- heterogeneous beliefs
- incomplete markets
- short sales constraints, and possibly ...
- (leverage) limits on an investor's ability to borrow in order to finance purchases of a risky asset

References

Prior to reading the following you might like to review our lectures on

- *Markov chains*
- *Asset pricing with finite state space*

Bubbles

Economists differ in how they define a *bubble*

The Harrison-Kreps model illustrates the following notion of a bubble that attracts many economists:

*A component of an asset price can be interpreted as a bubble when all investors agree that the current price of the asset exceeds what they believe the asset's underlying dividend stream justifies.*

4.7.2 Structure of the Model

The model simplifies by ignoring alterations in the distribution of wealth among investors having different beliefs about the fundamentals that determine asset payouts.

There is a fixed number $A$ of shares of an asset.

Each share entitles its owner to a stream of dividends $\{d_t\}$ governed by a Markov chain defined on a state space $S \in \{0, 1\}$.

The dividend obeys

$$d_t = \begin{cases} 
0 & \text{if } s_t = 0 \\
1 & \text{if } s_t = 1 
\end{cases}$$

The owner of a share at the beginning of time $t$ is entitled to the dividend paid at time $t$.

The owner of the share at the beginning of time $t$ is also entitled to sell the share to another investor during time $t$. 
Two types $h = a, b$ of investors differ only in their beliefs about a Markov transition matrix $P$ with typical element

$$P(i, j) = \mathbb{P}\{s_{t+1} = j \mid s_t = i\}$$

Investors of type $a$ believe the transition matrix

$$P_a = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

Investors of type $b$ think the transition matrix is

$$P_b = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

The stationary (i.e., invariant) distributions of these two matrices can be calculated as follows:

```julia
using QuantEcon
qa = [1/2 1/2; 2/3 1/3]
qb = [2/3 1/3; 1/4 3/4]
mcA = MarkovChain(qa)
mcB = MarkovChain(qb)
stationary_distributions(mcA)

1-element Array{Array{Float64,1},1}:
[0.571429, 0.428571]

stationary_distributions(mcB)

1-element Array{Array{Float64,1},1}:
[0.428571, 0.571429]
```

The stationary distribution of $P_a$ is approximately $\pi_A = [0.57, 0.43]$.

The stationary distribution of $P_b$ is approximately $\pi_B = [0.43, 0.57]$.

**Ownership Rights**

An owner of the asset at the end of time $t$ is entitled to the dividend at time $t + 1$ and also has the right to sell the asset at time $t + 1$.

Both types of investors are risk-neutral and both have the same fixed discount factor $\beta \in (0, 1)$.

In our numerical example, well set $\beta = 0.75$, just as Harrison and Kreps did.

Well eventually study the consequences of two different assumptions about the number of shares $A$ relative to the resources that our two types of investors can invest in the stock:

1. Both types of investors have enough resources (either wealth or the capacity to borrow) so that they can purchase the entire available stock of the asset\(^1\)

\(^1\) By assuming that both types of agent always have deep enough pockets to purchase all of the asset, the model takes wealth dynamics off the table. The Harrison-Kreps model generates high trading volume when the state changes either from 0 to 1 or from 1 to 0.
2. No single type of investor has sufficient resources to purchase the entire stock
Case 1 is the case studied in Harrison and Kreps
In case 2, both types of investor always hold at least some of the asset

Short Sales Prohibited

No short sales are allowed
This matters because it limits pessimists from expressing their opinions
- They can express their views by selling their shares
- They cannot express their pessimism more loudly by artificially manufacturing shares – that is, they cannot borrow shares from more optimistic investors and sell them immediately

Optimism and Pessimism

The above specifications of the perceived transition matrices $P_a$ and $P_b$, taken directly from Harrison and Kreps, build in stochastically alternating temporary optimism and pessimism
Remember that state 1 is the high dividend state
- In state 0, a type $a$ agent is more optimistic about next periods dividend than a type $b$ agent
- In state 1, a type $b$ agent is more optimistic about next periods dividend
However, the stationary distributions $\pi_A = [0.57, 0.43]$ and $\pi_B = [0.43, 0.57]$ tell us that a type $B$ person is more optimistic about the dividend process in the long run than is a type $A$ person
Transition matrices for the temporarily optimistic and pessimistic investors are constructed as follows
Temporarily optimistic investors (i.e., the investor with the most optimistic beliefs in each state) believe the transition matrix

$$
P_o = \begin{bmatrix}
\frac{1}{4} & \frac{1}{2} \\
\frac{1}{2} & \frac{1}{4}
\end{bmatrix}
$$

Temporarily pessimistic believe the transition matrix

$$
P_p = \begin{bmatrix}
\frac{1}{4} & \frac{1}{2} \\
\frac{1}{2} & \frac{1}{4}
\end{bmatrix}
$$

Well return to these matrices and their significance in the exercise

Information

Investors know a price function mapping the state $s_t$ at $t$ into the equilibrium price $p(s_t)$ that prevails in that state
This price function is endogenous and to be determined below
When investors choose whether to purchase or sell the asset at $t$, they also know $s_t$
4.7.3 Solving the Model

Now let's turn to solving the model.

This amounts to determining equilibrium prices under the different possible specifications of beliefs and constraints listed above.

In particular, we compare equilibrium price functions under the following alternative assumptions about beliefs:

1. There is only one type of agent, either \( a \) or \( b \).
2. There are two types of agent differentiated only by their beliefs. Each type of agent has sufficient resources to purchase all of the asset (Harrison and Krepss setting).
3. There are two types of agent with different beliefs, but because of limited wealth and/or limited leverage, both types of investors hold the asset each period.

Summary Table

The following table gives a summary of the findings obtained in the remainder of the lecture (you will be asked to recreate the table in an exercise).

It records implications of Harrison and Krepss specifications of \( P_a, P_b, \beta \)

\[
\begin{array}{|c|c|c|}
\hline
s_t & 0 & 1 \\
\hline
P_a & 1.33 & 1.22 \\
P_b & 1.45 & 1.91 \\
P_o & 1.85 & 2.08 \\
P_p & 1 & 1 \\
\tilde{P}_a & 1.85 & 1.69 \\
\tilde{P}_b & 1.69 & 2.08 \\
\hline
\end{array}
\]

Here

• \( P_a \) is the equilibrium price function under homogeneous beliefs \( P_a \)
• \( P_b \) is the equilibrium price function under homogeneous beliefs \( P_b \)
• \( P_o \) is the equilibrium price function under heterogeneous beliefs with optimistic marginal investors
• \( P_p \) is the equilibrium price function under heterogeneous beliefs with pessimistic marginal investors
• \( \tilde{P}_a \) is the amount type \( a \) investors are willing to pay for the asset
• \( \tilde{P}_b \) is the amount type \( b \) investors are willing to pay for the asset

We'll explain these values and how they are calculated one row at a time.

Single Belief Prices

Well start by pricing the asset under homogeneous beliefs.
(This is the case treated in the lecture on asset pricing with finite Markov states)

Suppose that there is only one type of investor, either of type $a$ or $b$, and that this investor always prices the asset

Let $p_h = \begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix}$ be the equilibrium price vector when all investors are of type $h$

The price today equals the expected discounted value of tomorrow's dividend and tomorrow's price of the asset:

$$p_h(s) = \beta (P_h(s, 0)(0 + p_h(0)) + P_h(s, 1)(1 + p_h(1))), \quad s = 0, 1$$

These equations imply that the equilibrium price vector is

$$\begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix} = \beta [I - \beta P_h]^{-1} P_h \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad (4.63)$$

The first two rows of the table report $p_a(s)$ and $p_b(s)$

Here's a function that can be used to compute these values

```julia
#=
Authors: Shunsuke Hori
=#
""
Function to Solve Single Beliefs
""
function price_single_beliefs(transition::Matrix, dividend_payoff::Vector; \beta::AbstractFloat=.75)
    # First compute inverse piece
    imbq_inv = inv(eye(size(transition, 1)) - \beta * transition)
    # Next compute prices
    prices = \beta * ((imbq_inv + transition) + dividend_payoff)
    return prices
end
```

**Single belief prices as benchmarks**

These equilibrium prices under homogeneous beliefs are important benchmarks for the subsequent analysis

- $p_h(s)$ tells what investor $h$ thinks is the fundamental value of the asset
- Here fundamental value means the expected discounted present value of future dividends

We will compare these fundamental values of the asset with equilibrium values when traders have different beliefs
Pricing under Heterogeneous Beliefs

There are several cases to consider

The first is when both types of agent have sufficient wealth to purchase all of the asset themselves

In this case the marginal investor who prices the asset is the more optimistic type, so that the equilibrium price $\bar{p}$ satisfies Harrison and Krepss key equation:

$$
\bar{p}(s) = \beta \max \left\{ P_a(s, 0)\bar{p}(0) + P_a(s, 1)(1 + \bar{p}(1)), \ P_b(s, 0)\bar{p}(0) + P_b(s, 1)(1 + \bar{p}(1)) \right\} \quad (4.64)
$$

for $s = 0, 1$

The marginal investor who prices the asset in state $s$ is of type $a$ if

$$
P_a(s, 0)\bar{p}(0) + P_a(s, 1)(1 + \bar{p}(1)) > P_b(s, 0)\bar{p}(0) + P_b(s, 1)(1 + \bar{p}(1))
$$

The marginal investor is of type $b$ if

$$
P_a(s, 1)\bar{p}(0) + P_a(s, 1)(1 + \bar{p}(1)) < P_b(s, 1)\bar{p}(0) + P_b(s, 1)(1 + \bar{p}(1))
$$

Thus the marginal investor is the (temporarily) optimistic type

Equation (4.64) is a functional equation that, like a Bellman equation, can be solved by

- starting with a guess for the price vector $\bar{p}$ and
- iterating to convergence on the operator that maps a guess $\bar{p}^j$ into an updated guess $\bar{p}^{j+1}$ defined by the right side of (4.64), namely

$$
\bar{p}^{j+1}(s) = \beta \max \left\{ P_a(s, 0)\bar{p}^j(0) + P_a(s, 1)(1 + \bar{p}^j(1)), \ P_b(s, 0)\bar{p}^j(0) + P_b(s, 1)(1 + \bar{p}^j(1)) \right\} \quad (4.65)
$$

for $s = 0, 1$

The third row of the table reports equilibrium prices that solve the functional equation when $\beta = .75$

Here the type that is optimistic about $s_{t+1}$ prices the asset in state $s_t$

It is instructive to compare these prices with the equilibrium prices for the homogeneous belief economies that solve under beliefs $P_a$ and $P_b$

Equilibrium prices $\bar{p}$ in the heterogeneous beliefs economy exceed what any prospective investor regards as the fundamental value of the asset in each possible state

Nevertheless, the economy recurrently visits a state that makes each investor want to purchase the asset for more than he believes its future dividends are worth

The reason is that he expects to have the option to sell the asset later to another investor who will value the asset more highly than he will

- Investors of type $a$ are willing to pay the following price for the asset

$$
\bar{p}_a(s) = \begin{cases} 
\bar{p}(0) & \text{if } s_t = 0 \\
\beta(P_a(1, 0)\bar{p}(0) + P_a(1, 1)(1 + \bar{p}(1))) & \text{if } s_t = 1
\end{cases}
$$
• Investors of type $b$ are willing to pay the following price for the asset

$$\hat{p}_b(s) = \begin{cases} 
\beta(P_b(0,0)\bar{p}(0) + P_b(0,1)(1 + \bar{p}(1))) & \text{if } s_t = 0 \\
\bar{p}(1) & \text{if } s_t = 1 
\end{cases}$$

Evidently, $\hat{p}_a(1) < \bar{p}(1)$ and $\hat{p}_b(0) < \bar{p}(0)$

Investors of type $a$ want to sell the asset in state 1 while investors of type $b$ want to sell it in state 0

• The asset changes hands whenever the state changes from 0 to 1 or from 1 to 0
• The valuations $\hat{p}_a(s)$ and $\hat{p}_b(s)$ are displayed in the fourth and fifth rows of the table
• Even the pessimistic investors who don’t buy the asset think that it is worth more than they think future dividends are worth

Here’s code to solve for $\bar{p}$, $\hat{p}_a$ and $\hat{p}_b$ using the iterative method described above:

```julia
"""
Function to Solve Optimistic Beliefs
"""

function price_optimistic_beliefs(transitions::Vector, dividend_payoff::Vector; β::AbstractFloat=.75, max_iter::Integer=50000, tol::AbstractFloat=1e-16)

    # We will guess an initial price vector of [0, 0]
p_new = [0,0]
p_old = [10.0,10.0]

    # We know this is a contraction mapping, so we can iterate to conv
    for i in 1:max_iter
        p_old = p_new
        temp = [maximum((q * p_old) + (q * dividend_payoff)) for q in transitions]
        p_new = β * temp

        # If we succeed in converging, break out of for loop
        if maximum(sqrt, ((p_new - p_old).^2)) < 1e-12
            break
        end

        temp=[minimum((q * p_old) + (q * dividend_payoff)) for q in transitions]
        ptwiddle = β * temp

        phat_a = [p_new[1], ptwiddle[2]]
        phat_b = [ptwiddle[1], p_new[2]]

    end

    return p_new, phat_a, phat_b
end
```

4.7. Asset Pricing III: Incomplete Markets 701
Insufficient Funds

Outcomes differ when the more optimistic type of investor has insufficient wealth or insufficient ability to borrow enough to hold the entire stock of the asset.

In this case, the asset price must adjust to attract pessimistic investors.

Instead of equation (4.64), the equilibrium price satisfies

\[
p(s) = \beta \min \{ P_a(s, 1)p(0) + P_a(s, 1)(1 + p(1)), P_b(s, 1)p(0) + P_b(s, 1)(1 + p(1)) \} \quad (4.66)
\]

and the marginal investor who prices the asset is always the one that values it less highly than does the other type.

Now the marginal investor is always the (temporarily) pessimistic type.

Notice from the sixth row of that the pessimistic price \( p \) is lower than the homogeneous belief prices \( p_a \) and \( p_b \) in both states.

When pessimistic investors price the asset according to (4.66), optimistic investors think that the asset is overpriced.

If they could, optimistic investors would willingly borrow at the one-period gross interest rate \( \beta^{-1} \) to purchase more of the asset.

Implicit constraints on leverage prohibit them from doing so.

When optimistic investors price the asset as in equation (4.64), pessimistic investors think that the asset is underpriced and would like to sell the asset short.

Constraints on short sales prevent that.

Here code to solve for \( \tilde{p} \) using iteration

```julia
# Function to Solve Pessimistic Beliefs
function price_pessimistic_beliefs(transitions::Vector, dividend_payoff::Vector; ::AbstractFloat=.75, max_iter::Integer=50000, tol::AbstractFloat=1e-16)
    # We will guess an initial price vector of [0, 0]
    p_new = [0.0, 0.0]
    p_old = [10.0, 10.0]

    # We know this is a contraction mapping, so we can iterate to converge
    for i in 1:max_iter
        p_old = p_new
        temp = [minimum((q * p_old) + (q * dividend_payoff)) for q in transitions]
        p_new = \beta * temp
        if norm(p_new - p_old) < tol
            break
        end
    end
    return p_new

    # If we succeeded in converging, break out of the loop
```

Chapter 4. Multiple Agent Models
if maximum(sqrt, ((p_new - p_old).^2)) < 1e-12
    break
end
return p_new
end

Further Interpretation

[Sch14] interprets the Harrison-Kreps model as a model of a bubble—a situation in which an asset price exceeds what every investor thinks is merited by the assets underlying dividend stream.

Scheinkman stresses these features of the Harrison-Kreps model:

- Compared to the homogeneous beliefs setting leading to the pricing formula, high volume occurs when the Harrison-Kreps pricing formula prevails.

Type $a$ investors sell the entire stock of the asset to type $b$ investors every time the state switches from $s_t = 0$ to $s_t = 1$.

Type $b$ investors sell the asset to type $a$ investors every time the state switches from $s_t = 1$ to $s_t = 0$.

Scheinkman takes this as a strength of the model because he observes high volume during famous bubbles.

- If the supply of the asset is increased sufficiently either physically (more houses are built) or artificially (ways are invented to short sell houses), bubbles end when the supply has grown enough to outstrip optimistic investors resources for purchasing the asset.

- If optimistic investors finance purchases by borrowing, tightening leverage constraints can extinguish a bubble.

Scheinkman extracts insights about effects of financial regulations on bubbles.

He emphasizes how limiting short sales and limiting leverage have opposite effects.

4.7.4 Exercises

Exercise 1

Recreate the summary table using the functions we have built above.

<table>
<thead>
<tr>
<th>$s_t$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_a$</td>
<td>1.33</td>
<td>1.22</td>
</tr>
<tr>
<td>$p_b$</td>
<td>1.45</td>
<td>1.91</td>
</tr>
<tr>
<td>$p_o$</td>
<td>1.85</td>
<td>2.08</td>
</tr>
<tr>
<td>$p_P$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\bar{p}_a$</td>
<td>1.85</td>
<td>1.69</td>
</tr>
<tr>
<td>$\bar{p}_b$</td>
<td>1.69</td>
<td>2.08</td>
</tr>
</tbody>
</table>
You will first need to define the transition matrices and dividend payoff vector

### 4.7.5 Solutions

#### Exercise 1

First we will obtain equilibrium price vectors with homogeneous beliefs, including when all investors are optimistic or pessimistic.

```julia
qa = [1/2 1/2; 2/3 1/3]  # Type a transition matrix
qb = [2/3 1/3; 1/4 3/4]  # Type b transition matrix
qopt = [1/2 1/2; 1/4 3/4]  # Optimistic investor transition matrix
qpess = [2/3 1/3; 2/3 1/3]  # Pessimistic investor transition matrix
dividendreturn = [0; 1]

transitions = [qa, qb, qopt, qpess]
labels = ["p_a", "p_b", "p_optimistic", "p_pessimistic"]

for (transition, label) in zip(transitions, labels)
    println(label)
    println(repeat("=" , 20))
    s0, s1 = round.(price_single_beliefs(transition, dividendreturn), 2)
    println("State 0: \$s0")
    println("State 1: \$s1")
    println(repeat("-", 20))
end
```

```
p_a
=========
State 0:  [ 1.33]
State 1:  [ 1.22]
--------

p_b
==========
State 0:  [ 1.45]
State 1:  [ 1.91]
--------

p_optimistic
=============
State 0:  [ 1.85]
State 1:  [ 2.08]
--------

p_pessimistic
==============
State 0:  [ 1.]
State 1:  [ 1.]
--------
```

We will use the `price_optimistic_beliefs` function to find the price under heterogeneous beliefs.
opt_beliefs = price_optimistic_beliefs([qa, qb], dividendreturn)
labels = ["p_optimistic", "p_hat_a", "p_hat_b"]

for (p, label) in zip(opt_beliefs, labels)
    println(label)
    println(repeat("=", 20))
    s0, s1 = round.(p, 2)
    println("State 0: $s0")
    println("State 1: $s1")
    println(repeat("-", 20))
end

p_optimistic
=============
State 0:  [ 1.85]
State 1:  [ 2.08]
-------------

p_hat_a
========
State 0:  [ 1.85]
State 1:  [ 1.69]
-------------

p_hat_b
========
State 0:  [ 1.69]
State 1:  [ 2.08]
-------------

Notice that the equilibrium price with heterogeneous beliefs is equal to the price under single beliefs with optimistic investors - this is due to the marginal investor being the temporarily optimistic type

4.8 Uncertainty Traps

4.8.1 Overview

In this lecture we study a simplified version of an uncertainty traps model of Fajgelbaum, Schaal and Taschereau-Dumouchel [FSTD15]

The model features self-reinforcing uncertainty that has big impacts on economic activity

In the model,

- Fundamentals vary stochastically and are not fully observable
- At any moment there are both active and inactive entrepreneurs; only active entrepreneurs produce
- Agents – active and inactive entrepreneurs – have beliefs about the fundamentals expressed as probability distributions
- Greater uncertainty means greater dispersions of these distributions
- Entrepreneurs are risk averse and hence less inclined to be active when uncertainty is high
• The output of active entrepreneurs is observable, supplying a noisy signal that helps everyone inside the model infer fundamentals

• Entrepreneurs update their beliefs about fundamentals using Bayes Law, implemented via Kalman filtering

Uncertainty traps emerge because:

• High uncertainty discourages entrepreneurs from becoming active

• A low level of participation – i.e., a smaller number of active entrepreneurs – diminishes the flow of information about fundamentals

• Less information translates to higher uncertainty, further discouraging entrepreneurs from choosing to be active, and so on

Uncertainty traps stem from a positive externality: high aggregate economic activity levels generates valuable information

4.8.2 The Model

The original model described in [FSTD15] has many interesting moving parts

Here we examine a simplified version that nonetheless captures many of the key ideas

Fundamentals

The evolution of the fundamental process \( \{ \theta_t \} \) is given by

\[
\theta_{t+1} = \rho \theta_t + \sigma_{\theta} w_{t+1}
\]

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 \( \tilde{M} \) of risk averse entrepreneurs

Output of the \( m \)-th entrepreneur, conditional on being active in the market at time \( t \), is equal to

\[
x_m = \theta + \epsilon_m \quad \text{where} \quad \epsilon_m \sim N(0, \gamma_x^{-1})
\]  

(4.67)

Here the time subscript has been dropped to simplify notation

The inverse of the shock variance, \( \gamma_x \), is called the shocks precision
The higher is the precision, the more informative $x_m$ is about the fundamental
Output shocks are independent across time and firms

**Information and Beliefs**

All entrepreneurs start with identical beliefs about $\theta_0$
Signals are publicly observable and hence all agents have identical beliefs always
Dropping time subscripts, beliefs for current $\theta$ are represented by the normal distribution $N(\mu, \gamma^{-1})$
Here $\gamma$ is the precision of beliefs; its inverse is the degree of uncertainty
These parameters are updated by Kalman filtering
Let

- $M \subset \{1, \ldots, M\}$ denote the set of currently active firms
- $M := |M|$ denote the number of currently active firms
- $X$ be the average output $\frac{1}{M} \sum_{m \in M} x_m$ of the active firms

With this notation and primes for next period values, we can write the updating of the mean and precision via

$$
\mu' = \rho \frac{\gamma \mu + M \gamma_x X}{\gamma + M \gamma_x} \quad (4.68)
$$

$$
\gamma' = \left( \frac{\rho^2}{\gamma + M \gamma_x} + \sigma_\theta^2 \right)^{-1} \quad (4.69)
$$

These are standard Kalman filtering results applied to the current setting
Exercise 1 provides more details on how (4.68) and (4.69) are derived, and then asks you to fill in remaining steps
The next figure plots the law of motion for the precision in (4.69) as a 45 degree diagram, with one curve for each $M \in \{0, \ldots, 6\}$
The other parameter values are $\rho = 0.99, \gamma_x = 0.5, \sigma_\theta = 0.5$
Points where the curves hit the 45 degree lines are long run steady states for precision for different values of $M$

Thus, if one of these values for $M$ remains fixed, a corresponding steady state is the equilibrium level of precision

- high values of $M$ correspond to greater information about the fundamental, and hence more precision in steady state
- low values of $M$ correspond to less information and more uncertainty in steady state

In practice, as well see, the number of active firms fluctuates stochastically
Participation

Omitting time subscripts once more, entrepreneurs enter the market in the current period if

\[ \mathbb{E}[u(x_m - F_m)] > c \]  \hspace{1cm} (4.70)

Here

- the mathematical expectation of \( x_m \) is based on (4.67) 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 (4.70)

The utility function has the constant absolute risk aversion form

\[ u(x) = \frac{1}{a} (1 - \exp(-ax)) \]  \hspace{1cm} (4.71)

where \( a \) is a positive parameter

Combining (4.70) and (4.71), entrepreneur \( m \) participates in the market (or is said to be active) when

\[ \frac{1}{a} \left\{ 1 - \mathbb{E}[\exp(-a(\theta + \epsilon_m - F_m))] \right\} > c \]

Using standard formulas for expectations of lognormal random variables, this is equivalent to the condition

\[ \psi(\mu, \gamma, F_m) := \frac{1}{a} \left( 1 - \exp \left( -a\mu + aF_m + \frac{a^2}{2} \left( \frac{1}{\gamma} + \frac{1}{\gamma_2} \right) \right) \right) - c > 0 \]  \hspace{1cm} (4.72)

4.8.3 Implementation

We want to simulate this economy

As a first step, lets put together a type that bundles

- the parameters, the current value of \( \theta \) and the current values of the two belief parameters \( \mu \) and \( \gamma \)
- methods to update \( \theta, \mu \) and \( \gamma \), as well as to determine the number of active firms and their outputs

The updating methods follow the laws of motion for \( \theta, \mu \) and \( \gamma \) given above

The method to evaluate the number of active firms generates \( F_1, \ldots, F_N \) and tests condition (4.72) for each firm

The function \texttt{UncertaintyTrapEcon} encodes as default values the parameters well use in the simulations below

4.8. Uncertainty Traps
mutable struct UncertaintyTrapEcon{TF<:AbstractFloat, TI<:Integer}
a::TF # Risk aversion
γ_x::TF # Production shock precision
ρ::TF # Correlation coefficient for θ
σ_θ::TF # Standard dev of θ shock
num_firms::TI # Number of firms
σ_F::TF # Std dev of fixed costs
c::TF # External opportunity cost
μ::TF # Initial value for μ
γ::TF # Initial value for γ
θ::TF # Initial value for θ
σ_x::TF # Standard deviation of shock
end

function UncertaintyTrapEcon(;a::AbstractFloat=1.5, γ_x::AbstractFloat=0.5, ρ::AbstractFloat=0.99, σ_θ::AbstractFloat=0.5, num_firms::Integer=100, σ_F::AbstractFloat=1.5, c::AbstractFloat=-420.0, μ_init::AbstractFloat=0.0, γ_init::AbstractFloat=4.0, θ_init::AbstractFloat=0.0)
    σ_x = sqrt(a / γ_x)
    UncertaintyTrapEcon(a, γ_x, ρ, σ_θ, num_firms, σ_F, c, μ_init, γ_init, θ_init, σ_x)
end

function ψ(uc::UncertaintyTrapEcon, F::Real)
    temp1 = -uc.a * (uc.μ - F)
    temp2 = 0.5 * uc.a^2 * (1 / uc.γ + 1 / uc.γ_x)
    return (1 / uc.a) * (1 - exp(temp1 + temp2)) - uc.c
end

# Update beliefs (μ, γ) based on aggregates X and M.

function update_beliefs!(uc::UncertaintyTrapEcon, X::Real, M::Real)
    γ_μ, ρ, σ_θ = uc.γ_x, uc.ρ, uc.σ_θ
    # Update μ
    temp1 = ρ * (uc.γ * uc.μ + M * γ_x * X)
    temp2 = uc.γ + M * γ_x
    uc.μ = temp1 / temp2
    # Update γ
    uc.γ = 1 / (ρ^2 / (uc.γ + M * γ_x) + σ_θ^2)
end

update_θ!(uc::UncertaintyTrapEcon, w::Real) = (uc.θ = uc.ρ * uc.θ + uc.σ_θ * w)
Generate aggregates based on current beliefs \((\mu, \gamma)\). This is a simulation step that depends on the draws for \(F\).

```julia
function gen_aggregates(uc::UncertaintyTrapEcon)
    F_vals = uc.\(\sigma_F \times \text{randn}(uc.\text{num}\_\text{firms})

    M = sum(\(\psi.(uc, F_\text{vals}) \times 0\) # Counts number of active firms
    \text{if } M > 0
        x_vals = uc.\(\theta + uc.\sigma_x \times \text{randn}(M)
        X = \text{mean}(x_vals)
    \text{else}
        X = 0.0
    \text{end}
    \text{return } X, M
end
```

In the results below we use this code to simulate time series for the major variables

### 4.8.4 Results

Let's look first at the dynamics of \(\mu\), which the agents use to track \(\theta\)

![Graph showing the tracking of \(\mu\) and \(\theta\) over time](image)

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.

### 4.8. Uncertainty Traps
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
4.8. Uncertainty Traps
Notice how the traps only take hold after a sequence of bad draws for the fundamental
Thus, the model gives us a propagation mechanism that maps bad random draws into long downturns in economic activity

4.8.5 Exercises

Exercise 1

Fill in the details behind (4.68) and (4.69) based on the following standard result (see, e.g., p. 24 of [YS05])

Fact Let \( x = (x_1, \ldots, x_M) \) be a vector of IID draws from common distribution \( N(\theta, 1/\gamma_x) \) and let \( \bar{x} \) be the sample mean. If \( \gamma_x \) is known and the prior for \( \theta \) is \( N(\mu, 1/\gamma) \), then the posterior distribution of \( \theta \) given \( x \) is

\[
\pi(\theta | x) = N(\mu_0, 1/\gamma_0)
\]

where

\[
\mu_0 = \frac{\mu \gamma + M \bar{x} \gamma_x}{\gamma + M \gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M \gamma_x
\]

Exercise 2

Modulo randomness, replicate the simulation figures shown above

- Use the parameter values listed as defaults in the function UncertaintyTrapEcon

4.8.6 Solutions

Exercise 1

This exercise asked you to validate the laws of motion for \( \gamma \) and \( \mu \) given in the lecture, based on the stated result about Bayesian updating in a scalar Gaussian setting
The stated result tells us that after observing average output \( X \) of the \( M \) firms, our posterior beliefs will be

\[ N(\mu_0, 1/\gamma_0) \]

where

\[
\mu_0 = \frac{\mu \gamma + M X \gamma_x}{\gamma + M \gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M \gamma_x
\]

If we take a random variable \( \theta \) with this distribution and then evaluate the distribution of \( \rho \theta + \sigma \theta w \) where \( w \) is independent and standard normal, we get the expressions for \( \mu' \) and \( \gamma' \) given in the lecture.
4.8.7 Exercise 2

First let's replicate the plot that illustrates the law of motion for precision, which is

$$\gamma_{t+1} = \left( \frac{\rho^2}{\gamma_t + M\gamma_x + \sigma^2} \right)^{-1}$$

Here $M$ is the number of active firms. The next figure plots $\gamma_{t+1}$ against $\gamma_t$ on a 45 degree diagram for different values of $M$.

```julia
using QuantEcon
using Gadfly, DataFrames, LaTeXStrings

econ = UncertaintyTrapEcon()
ρ, σ_θ, γ_x = econ.ρ, econ.σ_θ, econ.γ_x # simplify names

# grid for γ and γ_(t+1)
γ = linspace(1e-10, 3, 200)
M_range = 0:6
γp = 1 ./ (ρ^2 ./ (γ .+ γ_x .* M_range') + σ_θ^2)

p1 = plot(x=repeat(collect(γ), outer=[length(M_range)+1]),
         y=vec([γ γp]),
         color=repeat(["45 Degree"; map(string, M_range)], inner=[length(γ)]),
         Geom.line, Guide.colorkey(title="M"), Guide.xlabel("γ"), Guide.ylabel("γ' \to"))
```
The points where the curves hit the 45 degree lines are the long run steady states corresponding to each $M$, if that value of $M$ was to remain fixed. As the number of firms falls, so does the long run steady state of precision.

Next lets generate time series for beliefs and the aggregates – that is, the number of active firms and average output.

```julia
function QuantEcon.simulate(TF::AbstractFloat, TI::Integer)(
    uc::UncertaintyTrapEcon(TF, TI), capT::TI=2000)

    # allocate memory
    μ_vec = Vector{TF}(capT)
    θ_vec = Vector{TF}(capT)
    γ_vec = Vector{TF}(capT)
    X_vec = Vector{TF}(capT)
    M_vec = Vector{TI}(capT)

    # set initial using fields from object
    μ_vec[1] = uc.μ
    γ_vec[1] = uc.γ
    θ_vec[1] = 0
```

Chapter 4. Multiple Agent Models
```julia
# draw standard normal shocks
w_shocks = randn(capT)

for t=1:capT-1
    X, M = gen_aggregates(uc)
    X_vec[t] = X
    M_vec[t] = M
    update_beliefs!(uc, X, M)
    update_θ!(uc, w_shocks[t])
    μ_vec[t+1] = uc.μ
    γ_vec[t+1] = uc.γ
    θ_vec[t+1] = uc.θ
end

# Record final values of aggregates
X, M = gen_aggregates(uc)
X_vec[end] = X
M_vec[end] = M

return μ_vec, γ_vec, θ_vec, X_vec, M_vec
end
```

First let’s see how well μ tracks θ in these simulations

```julia
srand(42)  # set random seed for reproducible results
μ_vec, γ_vec, θ_vec, X_vec, M_vec = simulate(econ)
p2 = plot(x=repeat(collect(1:length(μ_vec)), outer=2)), y=[μ_vec; θ_vec],
   color=repeat(["μ", "θ"], inner=length(μ_vec)),
   Geom.line, Guide.colorkey(title="Variable"))
```

4.8. Uncertainty Traps
Now let's plot the whole thing together:

```julia
mdf = DataFrame(t=1:length(θ_vec), θ=θ_vec, μ=μ_vec, γ=γ_vec, M=M_vec)
p3 = plot(stack(mdf, collect(2:5)), x="t",
    ygroup="variable",
    y="value",
    Geom.subplot_grid(Geom.line, free_y_axis=true))
```
4.9 The Aiyagari Model

4.9.1 Overview

In this lecture we describe the structure of a class of models that build on work by Truman Bewley [Bew77]. We begin by discussing an example of a Bewley model due to Rao Aiyagari.

The model features

- Heterogeneous agents
- A single exogenous vehicle for borrowing and lending
- Limits on amounts individual agents may borrow

The Aiyagari model has been used to investigate many topics, including

- precautionary savings and the effect of liquidity constraints [Aiy94]
• risk sharing and asset pricing \[HL96\]
• the shape of the wealth distribution \[BBZ15\]
• etc., etc., etc.

References

The primary reference for this lecture is \[Aiy94\]
A textbook treatment is available in chapter 18 of \[LS18\]
A continuous time version of the model by SeHyoun Ahn and Benjamin Moll can be found here

4.9.2 The Economy

Households

Infinitely lived households / consumers face idiosyncratic income shocks
A unit interval of \textit{ex ante} identical households face a common borrowing constraint
The savings problem faced by a typical household is

\[
\max \mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)
\]

subject to

\[
a_{t+1} + c_t \leq wz_t + (1 + r)a_t \quad c_t \geq 0, \quad \text{and} \quad a_t \geq -B
\]

where

• \(c_t\) is current consumption
• \(a_t\) is assets
• \(z_t\) is an exogenous component of labor income capturing stochastic unemployment risk, etc.
• \(w\) is a wage rate
• \(r\) is a net interest rate
• \(B\) is the maximum amount that the agent is allowed to borrow

The exogenous process \(\{z_t\}\) follows a finite state Markov chain with given stochastic matrix \(P\)
The wage and interest rate are fixed over time
In this simple version of the model, households supply labor inelastically because they do not value leisure
4.9.3 Firms

Firms produce output by hiring capital and labor

Firms act competitively and face constant returns to scale

Since returns to scale are constant the number of firms does not matter

Hence we can consider a single (but nonetheless competitive) representative firm

The firms output is

\[ Y_t = AK_t^\alpha N^{1-\alpha} \]

where

- \( A \) and \( \alpha \) are parameters with \( A > 0 \) and \( \alpha \in (0, 1) \)
- \( K_t \) is aggregate capital
- \( N \) is total labor supply (which is constant in this simple version of the model)

The firms problem is

\[ \max_{K;N} \{ AK_t^\alpha N^{1-\alpha} - (r + \delta)K - wN \} \]

The parameter \( \delta \) is the depreciation rate

From the first-order condition with respect to capital, the firms inverse demand for capital is

\[ r = A\alpha \left( \frac{N}{K} \right)^{1-\alpha} - \delta \quad (4.73) \]

Using this expression and the firms first-order condition for labor, we can pin down the equilibrium wage rate as a function of \( r \) as

\[ w(r) = A(1 - \alpha)(A\alpha/(r + \delta))^{\alpha/(1-\alpha)} \quad (4.74) \]

**Equilibrium**

We construct a *stationary rational expectations equilibrium* (SREE)

In such an equilibrium

- prices induce behavior that generates aggregate quantities consistent with the prices
- aggregate quantities and prices are constant over time

In more detail, an SREE lists a set of prices, savings and production policies such that

- households want to choose the specified savings policies taking the prices as given
- firms maximize profits taking the same prices as given
the resulting aggregate quantities are consistent with the prices; in particular, the demand for capital equals the supply

aggregate quantities (defined as cross-sectional averages) are constant

In practice, once parameter values are set, we can check for an SREE by the following steps

1. pick a proposed quantity $K$ for aggregate capital
2. determine corresponding prices, with interest rate $r$ determined by (4.73) and a wage rate $w(r)$ as given in (4.74)
3. determine the common optimal savings policy of the households given these prices
4. compute aggregate capital as the mean of steady state capital given this savings policy

If this final quantity agrees with $K$ then we have a SREE

### 4.9.4 Code

Lets look at how we might compute such an equilibrium in practice

To solve the households dynamic programming problem well use the DiscreteDP type from QuantEcon.jl

Our first task is the least exciting one: write code that maps parameters for a household problem into the $R$ and $Q$ matrices needed to generate an instance of DiscreteDP

Below is a piece of boilerplate code that does just this

In reading the code, the following information will be helpful

- $R$ needs to be a matrix where $R[s, a]$ is the reward at state $s$ under action $a$
- $Q$ needs to be a three dimensional array where $Q[s, a, s']$ is the probability of transitioning to state $s'$ when the current state is $s$ and the current action is $a$

(For a detailed discussion of DiscreteDP see this lecture)

Here we take the state to be $s_t := (a_t, z_t)$, where $a_t$ is assets and $z_t$ is the shock

The action is the choice of next period asset level $a_{t+1}$

The type also includes a default set of parameters that well adopt unless otherwise specified

```julia
using QuantEcon

""
Stores all the parameters that define the household's problem.

#### Fields
- `r::Real` : interest rate
- `w::Real` : wage
- `σ::Real` : risk aversion
- `β::AbstractFloat` : discount factor
- `z_chain::MarkovChain` : MarkovChain for income
```
4.9. The Aiyagari Model

```
- `a_min::Real` : minimum on asset grid
- `a_max::Real` : maximum on asset grid
- `a_size::Integer` : number of points on asset grid
- `z_size::Integer` : number of points on income grid
- `n::Integer` : number of points in state space: (a, z)
- `s_vals::Array(TF) where TF<:AbstractFloat` : stores all the possible (a, z) combinations
- `s_i_vals::Array(TI) where TI<:Integer` : stores indices of all the possible (a, z) combinations
- `R::Array(TF)` where TF<:AbstractFloat` : reward array
- `Q::Array(TF)` where TF<:AbstractFloat` : transition probability array
- `u::Function` : utility function

mutable struct Household{TR<:Real, TF<:AbstractFloat, TI<:Integer}
    r::TR
    w::TR
    σ::TR
    β::TF
    z_chain::MarkovChain{TF, Array(TF, 2), Array(TF, 1)}
    a_min::TR
    a_max::TR
    a_size::TI
    a_vals::AbstractVector{TF}
    z_size::TI
    n::TI
    s_vals::Array(TF)
    s_i_vals::Array(TI)
    R::Array(TR)
    Q::Array(TR)
    u::Function
end

Constructor for `Household`

```

#### Arguments
- `r::Real(0.01)` : interest rate
- `w::Real(1.0)` : wage
- `β::AbstractFloat(0.96)` : discount factor
- `z_chain::MarkovChain` : MarkovChain for income
- `a_min::Real(1e-10)` : minimum on asset grid
- `a_max::Real(18.0)` : maximum on asset grid
- `a_size::TI(200)` : number of points on asset grid

```

function Household(TF<:AbstractFloat)(;
    r::Real=0.01,
    w::Real=1.0,
    σ::Real=1.0,
    β::TF=0.96,
    z_chain::MarkovChain(TF,Array(TF,2),Array(TF,1))
    =MarkovChain([0.9 0.1; 0.1 0.9], [0.1; 1.0]),
    a_min::Real=1e-10,
```
# set up grids
a_vals = linspace(a_min, a_max, a_size)
z_size = length(z_chain.state_values)
n = a_size*z_size
s_vals = gridmake(a_vals, z_chain.state_values)
s_i_vals = gridmake(1:a_size, 1:z_size)

# set up Q
Q = zeros(TF, n, a_size, n)
for next_s_i in 1:n
    for a_i in 1:a_size
        for s_i in 1:n
            z_i = s_i_vals[s_i, 2]
            next_z_i = s_i_vals[next_s_i, 2]
            next_a_i = s_i_vals[next_s_i, 1]
            if next_a_i == a_i
                Q[s_i, a_i, next_s_i] = z_chain.p[z_i, next_z_i]
            end
        end
    end
end

if σ == 1  # log utility
    u = x -> log(x)
else
    u = x -> (x^(1-σ)-1)/(1-σ)
end

# placeholder for R
R = fill(-Inf, n, a_size)
h = Household(r, w, σ, β, z_chain, a_min, a_max, a_size,
            a_vals, z_size, n, s_vals, s_i_vals, R, Q, u)

    setup_R!(h, r, w)

return h
end

Update the reward array of a Household object, given a new interest rate and wage.

#### Arguments
- `h::Household` : instance of Household type
- `r::Real(0.01)` : interest rate
- `w::Real(1.0)` : wage

```
As a first example of what we can do, let's compute and plot an optimal accumulation policy at fixed prices.

```
using LaTeXStrings
using Plots
pyplot()

# Example prices
r = 0.03
w = 0.956

# Create an instance of Household
am = Household(a_max=20.0, r=r, w=w)

# Use the instance to build a discrete dynamic program
am_ddp = DiscreteDP(am.R, am.Q, am.β)

# Solve using policy function iteration
results = solve(am_ddp, PFI)

# Simplify names
z_size, a_size = am.z_size, am.a_size
z_vals, a_vals = am.z_chain.state_values, am.a_vals
n = am.n

# Get all optimal actions across the set of a indices with z fixed in each column
a_star = reshape([a_vals[results.sigma[s_i]] for s_i in 1:n], a_size, z_size)

labels = [latexstring("z = $(z_vals[1])") latexstring("z = $(z_vals[2])")]
plot(a_vals, a_star, label=labels, lw=2, alpha=0.6)
plot!(a_vals, a_vals, label="", color=:black, linestyle=:dash)
plot!(xlabel="current assets", ylabel="next period assets", grid=false)
```
The plot shows asset accumulation policies at different values of the exogenous state.

Now we want to calculate the equilibrium.

Let's do this visually as a first pass.

The following code draws aggregate supply and demand curves.

The intersection gives equilibrium interest rates and capital.

```julia
# Firms' parameters
const A = 1
const N = 1
const α = 0.33
const β = 0.96
```
const \( \delta = 0.05 \)

""
Compute wage rate given an interest rate, \( r \)
""

function r_to_w(r :: Real)
    return A * (1 - \( \alpha \)) * (A * \( \alpha \) / \( r + \delta \)) ^ (\( \alpha \) / (1 - \( \alpha \)))
end

""
Inverse demand curve for capital. The interest rate associated with a given demand for capital \( K \).
""

function rd(K :: Real)
    return A * \( \alpha \) * (N / K) ^ (1 - \( \alpha \)) - \( \delta \)
end

""
Map prices to the induced level of capital stock.

#### Arguments
- `am::Household` : Household instance for problem we want to solve
- `r::Real` : interest rate

#### Returns
- The implied level of aggregate capital

function prices_to_capital_stock(am::Household, r::Real)
    # Set up problem
    w = r_to_w(r)
    setup_R!(am, r, w)
    aiyagari_ddp = DiscreteDP(am.R, am.Q, am.\( \beta \))
    # Compute the optimal policy
    results = solve(aiyagari_ddp, PFI)
    # Compute the stationary distribution
    stationary_probs = stationary_distributions(results.mc)[::, 1][1]
    # Return \( K \)
    return dot(am.s_vals[::, 1], stationary_probs)
end

# Create an instance of Household
am = Household(\( \beta=\beta \), a_max=20.0)

# Create a grid of r values at which to compute demand and supply of capital
num_points = 20
r_vals = linspace(0.005, 0.04, num_points)

# Compute supply of capital
k_vals = prices_to_capital_stock.(am, r_vals)
# Plot against demand for capital by firms

```julia
demand = rd.([k_vals])
labels = ["demand for capital" "supply of capital"]
plot(k_vals, [demand r_vals], label=labels, lw=2, alpha=0.6)
plot!(xlabel="capital", ylabel="interest rate", xlim=(2, 14), ylim=(0.0, 0.1))
```

Here is the corresponding plot

4.10 Default Risk and Income Fluctuations

Contents

- Default Risk and Income Fluctuations
  - Overview
  - Structure
  - Equilibrium
4.10.1 Overview

This lecture computes versions of Arellanos [Are08] model of sovereign default

The model describes interactions among default risk, output, and an equilibrium interest rate that includes a premium for endogenous default risk

The decision maker is a government of a small open economy that borrows from risk-neutral foreign creditors

The foreign lenders must be compensated for default risk

The government borrows and lends abroad in order to smooth the consumption of its citizens

The government repays its debt only if it wants to, but declining to pay has adverse consequences

The interest rate on government debt adjusts in response to the state-dependent default probability chosen by government

The model yields outcomes that help interpret sovereign default experiences, including

- countercyclical interest rates on sovereign debt
- countercyclical trade balances
- high volatility of consumption relative to output

Notably, long recessions caused by bad draws in the income process increase the government’s incentive to default

This can lead to

- spikes in interest rates
- temporary losses of access to international credit markets
- large drops in output, consumption, and welfare
- large capital outflows during recessions

Such dynamics are consistent with experiences of many countries

4.10.2 Structure

In this section we describe the main features of the model
Output, Consumption and Debt

A small open economy is endowed with an exogenous stochastically fluctuating potential output stream \( \{ y_t \} \)

Potential output is realized only in periods in which the government honors its sovereign debt

The output good can be traded or consumed

The sequence \( \{ y_t \} \) is described by a Markov process with stochastic density kernel \( p(y, y') \)

Households within the country are identical and rank stochastic consumption streams according to

\[
E \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{4.75}
\]

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 (4.75)

The government is the only domestic actor with access to foreign credit

Because households are averse to consumption fluctuations, the government will try to smooth consumption by borrowing from (and lending to) foreign creditors

Asset Markets

The only credit instrument available to the government is a one-period bond traded in international credit markets

The bond market has the following features

- The bond matures in one period and is not state contingent
- A purchase of a bond with face value \( B' \) is a claim to \( B' \) units of the consumption good next period
- To purchase \( B' \) next period costs \( qB' \) now, or, what is equivalent
- For selling \(-B'\) units of next period goods the seller earns \(-qB'\) of today's goods
  - if \( B' < 0 \), then \(-qB'\) units of the good are received in the current period, for a promise to repay \(-B'\) units next period
  - there is an equilibrium price function \( q(B', y) \) that makes \( q \) depend on both \( B' \) and \( y \)

Earnings on the government portfolio are distributed (or, if negative, taxed) lump sum to households

When the government is not excluded from financial markets, the one-period national budget constraint is
\[ c = y + B - q(B', y)B' \]  

(4.76)

Here and below, a prime denotes a next period value or a claim maturing next period. To rule out Ponzi schemes, we also require that \( B \geq -Z \) in every period:

- \( Z \) is chosen to be sufficiently large that the constraint never binds in equilibrium.

**Financial Markets**

Foreign creditors:

- are risk neutral
- know the domestic output stochastic process \( \{y_t\} \) and observe \( y_t, y_{t-1}, \ldots \), at time \( t \)
- can borrow or lend without limit in an international credit market at a constant international interest rate \( r \)
- receive full payment if the government chooses to pay
- receive zero if the government defaults on its one-period debt due

When a government is expected to default next period with probability \( \delta \), the expected value of a promise to pay one unit of consumption next period is \( 1 - \delta \).

Therefore, the discounted expected value of a promise to pay \( B \) next period is

\[ q = \frac{1 - \delta}{1 + r} \]  

(4.77)

Next we turn to how the government in effect chooses the default probability \( \delta \).

**Governments decisions**

At each point in time \( t \), the government chooses between

1. defaulting
2. meeting its current obligations and purchasing or selling an optimal quantity of one-period sovereign debt

Defaulting means declining to repay all of its current obligations.

If the government defaults in the current period, then consumption equals current output. But a sovereign default has two consequences:

1. Output immediately falls from \( y \) to \( h(y) \), where \( 0 \leq h(y) \leq y \):
   - it returns to \( y \) only after the country regains access to international credit markets
2. The country loses access to foreign credit markets
Reentering international credit market

While in a state of default, the economy regains access to foreign credit in each subsequent period with probability $\theta$

4.10.3 Equilibrium

Informally, an equilibrium is a sequence of interest rates on its sovereign debt, a stochastic sequence of government default decisions and an implied flow of household consumption such that

1. Consumption and assets satisfy the national budget constraint
2. The government maximizes household utility taking into account
   • the resource constraint
   • the effect of its choices on the price of bonds
   • consequences of defaulting now for future net output and future borrowing and lending opportunities
3. The interest rate on the governments debt includes a risk-premium sufficient to make foreign creditors expect on average to earn the constant risk-free international interest rate

To express these ideas more precisely, consider first the choices of the government, which

1. enters a period with initial assets $B$, or what is the same thing, initial debt to be repaid now of $-B$
2. observes current output $y$, and
3. chooses either
   (a) to default, or
   (b) to pay $-B$ and set next periods debt due to $-B'$

In a recursive formulation,

- state variables for the government comprise the pair $(B, y)$
- $v(B, y)$ is the optimum value of the governments problem when at the beginning of a period it faces the choice of whether to honor or default
- $v_c(B, y)$ is the value of choosing to pay obligations falling due
- $v_d(y)$ is the value of choosing to default

$v_d(y)$ does not depend on $B$ because, when access to credit is eventually regained, net foreign assets equal 0.

Expressed recursively, the value of defaulting is

$$v_d(y) = u(h(y)) + \beta \int \left\{ \theta v(0, y') + (1 - \theta)v_d(y') \right\} p(y, y') dy'$$

The value of paying is

$$v_c(B, y) = \max_{B' \geq -Z} \left\{ u(y - q(B', y)B' + B) + \beta \int v(B', y') p(y, y') dy' \right\}$$
The three value functions are linked by

\[ v(B, y) = \max \{ v_c(B, y), v_d(y) \} \]

The government chooses to default when

\[ v_c(B, y) < v_d(y) \]

and hence given \( B' \) the probability of default next period is

\[ \delta(B', y) := \int \mathbb{I}\{v_c(B', y') < v_d(y')\} p(y, y') dy' \]  \hspace{1cm} (4.78)

Given zero profits for foreign creditors in equilibrium, we can combine (4.77) and (4.78) to pin down the bond price function:

\[ q(B', y) = \frac{1 - \delta(B', y)}{1 + r} \]  \hspace{1cm} (4.79)

**Definition of equilibrium**

An *equilibrium* is

- a pricing function \( q(B', y) \),
- a triple of value functions \( (v_c(B, y), v_d(y), v(B, y)) \),
- a decision rule telling the government when to default and when to pay as a function of the state \( (B, y) \), and
- an asset accumulation rule that, conditional on choosing not to default, maps \( (B, y) \) into \( B' \)

such that

- The three Bellman equations for \( (v_c(B, y), v_d(y), v(B, y)) \) are satisfied
- Given the price function \( q(B', y) \), the default decision rule and the asset accumulation decision rule attain the optimal value function \( v(B, y) \), and
- The price function \( q(B', y) \) satisfies equation (4.79)

### 4.10.4 Computation

Let's now compute an equilibrium of Arellanos model

The equilibrium objects are the value function \( v(B, y) \), the associated default decision rule, and the pricing function \( q(B', y) \)

We will use our code to replicate Arellanos results

After that we will perform some additional simulations
The majority of the code below was written by Chase Coleman

It uses a slightly modified version of the algorithm recommended by Arellano

• The appendix to [Are08] recommends value function iteration until convergence, updating the price, and then repeating

• Instead, we update the bond price at every value function iteration step

The second approach is faster and the two different procedures deliver very similar results

Here is a more detailed description of our algorithm:

1. Guess a value function \( v(B, y) \) and price function \( q(B', y) \)

2. At each pair \( (B, y) \),
   • update the value of defaulting \( v_d(y) \)
   • update the value of continuing \( v_c(B, y) \)

3. Update the value function \( v(B, y) \), the default rule, the implied ex ante default probability, and the price function

4. Check for convergence. If converged, stop. If not, go to step 2.

We use simple discretization on a grid of asset holdings and income levels

The output process is discretized using Tauchen quadrature method

The code can be found in the file arellano_vfi.jl but we repeat it here for convenience

(Result and discussion follow the code)

```julia
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
* `\beta::AbstractFloat`: Time discounting parameter
* `\gamma::AbstractFloat`: Risk aversion parameter
* `\rho::AbstractFloat`: World interest rate
* `\mu::AbstractFloat`: Autoregressive coefficient on income process
* `\eta::AbstractFloat`: Standard deviation of noise in income process
* `\theta::AbstractFloat`: Probability of re-entering the world financial sector after default
* `\eta::Integer`: Number of points to use in approximation of income process
* `\lambda::Integer`: Number of points to use in approximation of asset holdings
* `\gamma::Vector{AbstractFloat}`: This is the grid used to approximate income process
```

Chapter 4. Multiple Agent Models
4.10. Default Risk and Income Fluctuations

```julia
# Model Parameters
β::TF
γ::TF
r::TF
ρ::TF
η::TF
θ::TF

# Grid Parameters
ny::TI
nB::TI
ygrid::Vector[TF]
ydefgrid::Vector[TF]
Bgrid::Vector[TF]
Π::Matrix[TF]

# Value function
vf::Matrix[TF]
vdf::Matrix[TF]
vc::Matrix[TF]
policy::Matrix[TF]
q::Matrix[TF]
defprob::Matrix[TF]

end
```

This is the default constructor for building an economy as presented in Arellano 2008.

#### Arguments

- `β::AbstractFloat(0.953)`: Time discounting parameter
- `γ::AbstractFloat(2.0)`: Risk aversion parameter
- `r::AbstractFloat(0.017)`: World interest rate
- `ρ::AbstractFloat(0.945)`: Autoregressive coefficient on income process
* `\eta`: AbstractFloat(0.025)`: Standard deviation of noise in income process
* `\theta`: AbstractFloat(0.282)`: Probability of re-entering the world financial sector after default
* `ny`: Integer(21)`: Number of points to use in approximation of income process
* `nB`: Integer(251)`: Number of points to use in approximation of asset holdings

```julia
function ArellanoEconomy{TF<:AbstractFloat}(\beta::TF=.953, \gamma::TF=2., r::TF=0.017,
    \rho::TF=0.945, \eta::TF=0.025, \theta::TF=0.282,
    ny::Integer=21, nB::Integer=251)

    # Create grids
    Bgrid = collect(linspace(.4, .4, nB))
    mc = tauchen(ny, \rho, \eta)
    \Pi = mc.p
    ygrid = exp.(mc.state_values)
    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{TF}(nB, ny)
    q = ones(nB, ny) .* (1 / (1 + r))
    defprob = Array{TF}(nB, ny)

    return ArellanoEconomy(\beta, \gamma, r, \rho, \eta, \theta, ny, nB, ygrid, ydefgrid, Bgrid, \Pi,
        vf, vd, vc, policy, q, defprob)
end
```

```julia
u(ae::ArellanoEconomy, c) = c^(1 - ae.\gamma) / (1 - ae.\gamma)
```

```julia
unpack(ae::ArellanoEconomy) =
    ae.\beta, ae.\gamma, ae.r, ae.\rho, ae.\eta, ae.ny, ae.nB
unpackgrids(ae::ArellanoEconomy) =
    ae.ygrid, ae.ydefgrid, ae.Bgrid, ae.\Pi, ae.vf, ae.vd, ae.vc, ae.policy, ae.
```

```julia
# 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
```
function one_step_update!(TF <: AbstractFloat)(ae::ArellanoEconomy, EV::Matrix, EVd::Matrix, EVc::Matrix)

    # Unpack stuff
    β, γ, 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
        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

#Unpack 4.10. Default Risk and Income Fluctuations 737
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, 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 + ρ))

    Void
end
```

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, 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 \( \Pi \)' because of order value function dimensions)
copy!(V_upd, ae.vf)
EV = ae.vf * \Pi_t
EVd = ae.vd * \Pi_t
EVc = ae.vc * \Pi_t

    # Update Value Function
one_step_update!(ae, EV, EVd, EVc)

    # Update prices
compute_prices!(ae)

dist = maximum(abs, V_upd - ae.vf)

    if it%25 == 0
        println("Finished iteration \$(it) with dist of \$(dist)")
    end

end

Void

This function simulates the Arellano economy

#### Arguments

- `ae::ArellanoEconomy`: This is the economy we would like to solve
- `capT::Integer`: Number of periods to simulate
- `\(y_{init}\)::AbstractFloat(mean(ae.ygrid)`: The level of income we would like to start with
- `\(B_{init}\)::AbstractFloat(mean(ae.Bgrid)`: The level of asset holdings we would like to start with

#### Returns

- `\(B_{sim\_val}\)::Vector{TI}`: Simulated values of assets
- `\(y_{sim\_val}\)::Vector{TF}`: Simulated values of income
- `\(q_{sim\_val}\)::Vector{TF}`: Simulated values of prices
- `\(\text{default\_status}\)::Vector{Bool}`: Simulated default status (true if in default)
# Notes

* This updates all value functions, policy functions, and prices in place.

```julia
function QuantEcon.simulate( TI::Integer, TF::AbstractFloat )
    ae::ArellanoEconomy,
        capT::TI=5000;
        y_init::TF=mean(ae.ygrid),
        B_init::TF=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)

            # Create a QE MarkovChain
            mc = MarkovChain(ae.II)
            y_sim_indices = simulate(mc, capT+1; init=y_init_ind)

            # Allocate and Fill output
            y_sim_val = Vector(TF)(capT+1)
            B_sim_val, q_sim_val = similar(y_sim_val), similar(y_sim_val)
            B_sim_indices = Vector(TI)(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
                        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
                end
```
```julia
# 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 θ exit default status
    if rand() < ae.θ
        default_status[t+1] = false
    else
        default_status[t+1] = true
    end
end

return (y_sim_val[1:capT], B_sim_val[1:capT], q_sim_val[1:capT],
        default_status[1:capT])
end
```

### 4.10.5 Results

Lets start by trying to replicate the results obtained in [Are08]

In what follows, all results are computed using Arellanos parameter values

The values can be seen in the function `ArelanoEconomy` shown above

- For example, $r=0.017$ matches the average quarterly rate on a 5 year US treasury over the period 1983–2001

Details on how to compute the figures are reported as solutions to the exercises

The first figure shows the bond price schedule and replicates Figure 3 of Arellano, where $y_L$ and $Y_H$ are particular below average and above average values of output $y$
• $y_L$ is 5% below the mean of the $y$ grid values
• $y_H$ is 5% above the mean of the $y$ grid values

The grid used to compute this figure was relatively coarse ($ny, nB = 21, 251$) in order to match Arrelanos findings.

Here’s the same relationships computed on a finer grid ($ny, nB = 51, 551$).
In either case, the figure shows that

- Higher levels of debt (larger $-B'$) induce larger discounts on the face value, which correspond to higher interest rates
- Lower income also causes more discounting, as foreign creditors anticipate greater likelihood of default

The next figure plots value functions and replicates the right hand panel of Figure 4 of [Are08]
We can use the results of the computation to study the default probability \( \delta(B', y) \) defined in (4.78).

The next plot shows these default probabilities over \((B', y)\) as a heat map.
As anticipated, the probability that the government chooses to default in the following period increases with indebtedness and falls with income.

Next let's run a time series simulation of \( y_t \), \( B_t \) and \( q(B_{t+1}, y_t) \).

The grey vertical bars correspond to periods when the economy is excluded from financial markets because of a past default.
One notable feature of the simulated data is the nonlinear response of interest rates. Periods of relative stability are followed by sharp spikes in the discount rate on government debt.
4.10.6 Exercises

Exercise 1

To the extent that you can, replicate the figures shown above
- Use the parameter values listed as defaults in the function \textit{ArellanoEconomy}
- The time series will of course vary depending on the shock draws

4.10.7 Solutions

```julia
using Gadfly, Compose, ColorTypes, DataFrames

Compute the value function, policy and equilibrium prices

\[
\begin{align*}
\text{ae} &= \text{ArellanoEconomy}(\beta=0.953, \quad \text{# time discount rate} \\
\gamma=2., \quad \text{# risk aversion} \\
r=0.017, \quad \text{# international interest rate} \\
rho=0.945, \quad \text{# persistence in output} \\
\eta=0.025, \quad \text{# st dev of output shock} \\
\theta=0.282, \quad \text{# prob of regaining access} \\
n_y=21, \quad \text{# number of points in y grid} \\
n_B=251) \quad \text{# number of points in B grid}
\end{align*}
\]

# now solve the model on the grid.
\text{vfi!}!(\text{ae})

Finished iteration 25 with dist of 0.3424484168091375
Finished iteration 50 with dist of 0.098203940728843
Finished iteration 75 with dist of 0.029158637651295
Finished iteration 100 with dist of 0.00261840093811827
Finished iteration 125 with dist of 0.000785770921179818
Finished iteration 150 with dist of 0.000235832687651283
Finished iteration 175 with dist of 0.00005832460845048
Finished iteration 200 with dist of 7.07819565413636e-5
Finished iteration 225 with dist of 2.12438876549561e-5
Finished iteration 250 with dist of 6.37626793320578e-6
Finished iteration 275 with dist of 1.91376851657833e-6
Finished iteration 300 with dist of 5.74396178208729e-7
Finished iteration 325 with dist of 1.72398735287516e-7
Finished iteration 350 with dist of 5.1743049563026e-8
Finished iteration 375 with dist of 1.55302863902306e-8

Compute the bond price schedule as seen in figure 3 of Arellano (2008)

\[
\begin{align*}
\text{high, low} &= \text{mean}!(\text{ae}.ygrid) \times 1.05, \text{mean}!(\text{ae}.ygrid) \times 0.95 \\
iy\_high, iy\_low &= \text{map}(x \rightarrow \text{searchsortedfirst}(\text{ae}.ygrid, x), (\text{high, low}))
\end{align*}
\]

# Extract a suitable plot grid

4.10. Default Risk and Income Fluctuations 747
Draw a plot of the value functions
Draw a heat map for default probability

```
p2=plot(x=repeat(ae.Bgrid, outer=[2]),
       y=vec(ae.xf[; iy_low, iy_high]),
       color=repeat([:Low, :High], inner=[length(ae.Bgrid)]),
       Guide.title("Value functions"),
       Guide.xlabel("B"), Guide.ylabel("V(y,B)"),
       Guide.colorkey("y"), Geom.line)
```

```
p3=plot(x_min=repeat(ae.Bgrid[1:end-1], inner=[ae.ny-1]),
        x_max=repeat(ae.Bgrid[2:end], inner=[ae.ny-1]),
        y_min=repeat(ae.ygrid[1:end-1], outer=[ae.nB-1]),
        y_max=repeat(ae.ygrid[2:end], outer=[ae.nB-1]),
        x=(repeat(ae.Bgrid[1:end-1], inner=[ae.ny-1]) + repeat(ae.Bgrid[2:end],
              inner=[ae.ny-1]))/2,
        y=(repeat(ae.ygrid[1:end-1], outer=[ae.nB-1]) + repeat(ae.ygrid[2:end],
              outer=[ae.nB-1]))/2,
        color=clamp.(vec(ae.defprob[1:end-1, 1:end-1]'), 0, 1),
        Geom.rectbin,
        Guide.xlabel("B"), Guide.ylabel("y"))
```
Plot a time series of major variables simulated from the model

```julia
# set random seed for consistent result
srand(348938)

# simulate
T = 250
y_vec, B_vec, q_vec, default_vec = simulate(ae, T)

# find starting and ending periods of recessions
defs = find(default_vec)
def_breaks = diff(defs) .> 1
def_start = defs[[true; def_breaks]]
def_end = defs[[def_breaks; true]]

# construct boxes that shade periods of default
def_box = Guide.annotation(compose(context(),
    [rectangle(i[1], 0h, i[2]-i[1], 1h)
     for i=zip(def_start, def_end)]...,
    fill(RGBA(0.5, 0.5, 0.5, 0.2))))

# xy labels are common for all plots
xy_lab = [Guide.xlabel("time"), Guide.ylabel("")]
```
# now iterate over three variables and put them into an array
p4 = Gadfly.Plot[]
for (vec, name) in [(y_vec, "Output"), (B_vec, "Foreign assets"), (q_vec, "Bond price")]
    push!(p4,
        plot(x=1:T, y=vec, Geom.line, def_box, Guide.title(name), xy_lab...)
    )
end

# set final plot height and vertically stack the above three plots
set_default_plot_size(6inch, 8inch)
vstack(p4...)
4.11 Globalization and Cycles

This lecture is coauthored with Chase Coleman

4.11.1 Overview

In this lecture, we review the paper Globalization and Synchronization of Innovation Cycles by Kiminori Matsuyama, Laura Gardini and Iryna Sushko.

This model helps us understand several interesting stylized facts about the world economy.

One of these is synchronized business cycles across different countries.

Most existing models that generate synchronized business cycles do so by assumption, since they tie output in each country to a common shock.

They also fail to explain certain features of the data, such as the fact that the degree of synchronization tends to increase with trade ties.
By contrast, in the model we consider in this lecture, synchronization is both endogenous and increasing with the extent of trade integration.

In particular, as trade costs fall and international competition increases, innovation incentives become aligned and countries synchronize their innovation cycles.

**Background**

The model builds on work by Judd [Jud85], Deneckner and Judd [DJ92] and Helpman and Krugman [HK85] by developing a two-country model with trade and innovation.

On the technical side, the paper introduces the concept of coupled oscillators to economic modeling.

As we will see, coupled oscillators arise endogenously within the model.

Below we review the model and replicate some of the results on synchronization of innovation across countries.

### 4.11.2 Key Ideas

It is helpful to begin with an overview of the mechanism.

**Innovation Cycles**

As discussed above, two countries produce and trade with each other.

In each country, firms innovate, producing new varieties of goods and, in doing so, receiving temporary monopoly power.

Imitators follow and, after one period of monopoly, what had previously been new varieties now enter competitive production.

Firms have incentives to innovate and produce new goods when the mass of varieties of goods currently in production is relatively low.

In addition, there are strategic complementarities in the timing of innovation.

Firms have incentives to innovate in the same period, so as to avoid competing with substitutes that are competitively produced.

This leads to temporal clustering in innovations in each country.

After a burst of innovation, the mass of goods currently in production increases.

However, goods also become obsolete, so that not all survive from period to period.

This mechanism generates a cycle, where the mass of varieties increases through simultaneous innovation and then falls through obsolescence.
Synchronization

In the absence of trade, the timing of innovation cycles in each country is decoupled. This will be the case when trade costs are prohibitively high. If trade costs fall, then goods produced in each country penetrate each other's markets. As illustrated below, this leads to synchronization of business cycles across the two countries.

4.11.3 Model

Let's write down the model more formally.

(The treatment is relatively terse since full details can be found in the original paper.)

Time is discrete with \( t = 0, 1, \ldots \).

There are two countries indexed by \( j \) or \( k \).

In each country, a representative household inelastically supplies \( L_j \) units of labor at wage rate \( w_{j,t} \).

Without loss of generality, it is assumed that \( L_1 \geq L_2 \).

Households consume a single nontradeable final good which is produced competitively.

Its production involves combining two types of tradeable intermediate inputs via

\[
Y_{k,t} = C_{k,t} = \left( \frac{X^o_{k,t}}{1-\alpha} \right)^{1-\alpha} \left( \frac{X_{k,t}}{\alpha} \right)^\alpha
\]

Here \( X^o_{k,t} \) is a homogeneous input which can be produced from labor using a linear, one-for-one technology. It is freely tradeable, competitively supplied, and homogeneous across countries. By choosing the price of this good as numeraire and assuming both countries find it optimal to always produce the homogeneous good, we can set \( w_{1,t} = w_{2,t} = 1 \).

The good \( X_{k,t} \) is a composite, built from many differentiated goods via

\[
X_{k,t}^{1-\frac{1}{\sigma}} = \int_{\Omega_t} [x_{k,t}(\nu)]^{1-\frac{1}{\sigma}} d\nu
\]

Here \( x_{k,t}(\nu) \) is the total amount of a differentiated good \( \nu \in \Omega_t \) that is produced. The parameter \( \sigma > 1 \) is the direct partial elasticity of substitution between a pair of varieties and \( \Omega_t \) is the set of varieties available in period \( t \).

We can split the varieties into those which are supplied competitively and those supplied monopolistically; that is, \( \Omega_t = \Omega^c_t + \Omega^m_t \).
Prices

Demand for differentiated inputs is

\[ x_{k,t}(\nu) = \left( \frac{p_{k,t}(\nu)}{P_{k,t}} \right)^{-\sigma} \alpha L_k P_{k,t} \]

Here

- \( p_{k,t}(\nu) \) is the price of the variety \( \nu \) and
- \( P_{k,t} \) is the price index for differentiated inputs in \( k \), defined by

\[ [P_{k,t}]^{1-\sigma} = \int_{\Omega_t} [p_{k,t}(\nu)]^{1-\sigma} d\nu \]

The price of a variety also depends on the origin, \( j \), and destination, \( k \), of the goods because shipping varieties between countries incurs an iceberg trade cost \( \tau_{j,k} \).

Thus the effective price in country \( k \) of a variety \( \nu \) produced in country \( j \) becomes \( p_{k,t}(\nu) = \tau_{j,k} p_{j,t}(\nu) \).

Using these expressions, we can derive the total demand for each variety, which is

\[ D_{j,t}(\nu) = \sum_k \tau_{j,k} x_{k,t}(\nu) = \alpha A_{j,t}(p_{j,t}(\nu))^{-\sigma} \]

where

\[ A_{j,t} := \sum_k \frac{\rho_{j,k} L_k}{(P_{k,t})^{1-\sigma}} \quad \text{and} \quad \rho_{j,k} = (\tau_{j,k})^{1-\sigma} \leq 1 \]

It is assumed that \( \tau_{1,1} = \tau_{2,2} = 1 \) and \( \tau_{1,2} = \tau_{2,1} = \tau \) for some \( \tau > 1 \), so that

\[ \rho_{1,2} = \rho_{2,1} = \rho := \tau^{1-\sigma} < 1 \]

The value \( \rho \in [0, 1) \) is a proxy for the degree of globalization.

Producing one unit of each differentiated variety requires \( \psi \) units of labor, so the marginal cost is equal to \( \psi \) for \( \nu \in \Omega_{j,t} \).

Additionally, all competitive varieties will have the same price (because of equal marginal cost), which means that, for all \( \nu \in \Omega^c \),

\[ p_{j,t}(\nu) = p_{j,t}^c := \psi \quad \text{and} \quad D_{j,t} = y_{j,t}^c := \alpha A_{j,t}(p_{j,t}^c)^{-\sigma} \]

Monopolists will have the same marked-up price, so, for all \( \nu \in \Omega^m \),

\[ p_{j,t}(\nu) = p_{j,t}^m := \frac{\psi}{1 - \frac{1}{\sigma}} \quad \text{and} \quad D_{j,t} = y_{j,t}^m := \alpha A_{j,t}(p_{j,t}^m)^{-\sigma} \]

Define

\[ \theta := \frac{p_{j,t}^c y_{j,t}^c}{p_{j,t}^m y_{j,t}^m} = \left( 1 - \frac{1}{\sigma} \right)^{1-\sigma} \]

Using the preceding definitions and some algebra, the price indices can now be rewritten as

\[ \left( \frac{P_{k,t}}{\psi} \right)^{1-\sigma} = M_{k,t} + \rho M_{j,t} \quad \text{where} \quad M_{j,t} := N_{j,t}^c + \frac{N_{j,t}^m}{\theta} \]

The symbols \( N_{j,t}^c \) and \( N_{j,t}^m \) will denote the measures of \( \Omega^c \) and \( \Omega^m \) respectively.

4.11. Globalization and Cycles
New Varieties

To introduce a new variety, a firm must hire $f$ units of labor per variety in each country

Monopolist profits must be less than or equal to zero in expectation, so

$$N_{j,t}^m \geq 0, \quad \pi_{j,t}^m := (p_{j,t}^m - \psi)y_{j,t}^m - f \leq 0 \quad \text{and} \quad \pi_{j,t}^m N_{j,t}^m = 0$$

With further manipulations, this becomes

$$N_{j,t}^m = \theta(M_{j,t} - N_{j,t}^c) \geq 0, \quad \frac{1}{\sigma} \left[ \frac{\alpha L_j}{\theta(M_{j,t} + \rho M_{k,t})} + \frac{\alpha L_k}{\theta(M_{j,t} + M_{k,t}/\rho)} \right] \leq f$$

Law of Motion

With $\delta$ as the exogenous probability of a variety becoming obsolete, the dynamic equation for the measure of firms becomes

$$N_{j,t+1}^c = \delta(N_{j,t}^c + N_{j,t}^m) = \delta(N_{j,t}^c + \theta(M_{j,t} - N_{j,t}^c))$$

We will work with a normalized measure of varieties

$$n_{j,t} := \frac{\theta f N_{j,t}^c}{\alpha (L_1 + L_2)}, \quad i_{j,t} := \frac{\theta f N_{j,t}^m}{\alpha (L_1 + L_2)}, \quad m_{j,t} := \frac{\theta f M_{j,t}}{\alpha (L_1 + L_2)} = n_{j,t} + \frac{i_{j,t}}{\theta}$$

We also use $s_j := \frac{L_j}{L_1 + L_2}$ to be the share of labor employed in country $j$

We can use these definitions and the preceding expressions to obtain a law of motion for $n_t := (n_{1,t}, n_{2,t})$

In particular, given an initial condition, $n_0 = (n_{1,0}, n_{2,0}) \in \mathbb{R}_+^2$, the equilibrium trajectory, $\{n_t\}_t=0^\infty = \{(n_{1,t}, n_{2,t})\}_t=0^\infty$, is obtained by iterating on $n_{t+1} = F(n_t)$ where $F : \mathbb{R}_+^2 \to \mathbb{R}_+^2$ is given by

$$F(n_t) = \begin{cases} 
    (\delta(\theta s_1(\rho) + (1-\theta)n_{1,t}), \delta(\theta s_2(\rho) + (1-\theta)n_{2,t})) & \text{for } n_t \in D_{LL} \\
    (\delta n_{1,t}, \delta n_{2,t}) & \text{for } n_t \in D_{HH} \\
    (\delta n_{1,t}, \delta(\theta h_2(n_{1,t}) + (1-\theta)n_{2,t})) & \text{for } n_t \in D_{HL} \\
    (\delta(\theta h_1(n_{2,t}) + (1-\theta)n_{1,t}), \delta n_{2,t}) & \text{for } n_t \in D_{LH}
\end{cases}$$

Here

$$D_{LL} := \{(n_1, n_2) \in \mathbb{R}_+^2 | n_1 \leq s_j(\rho)\}$$

$$D_{HH} := \{(n_1, n_2) \in \mathbb{R}_+^2 | n_2 \geq h_j(\rho)\}$$

$$D_{HL} := \{(n_1, n_2) \in \mathbb{R}_+^2 | n_1 \geq s_1(\rho) \text{ and } n_2 \leq h_2(n_1)\}$$

$$D_{LH} := \{(n_1, n_2) \in \mathbb{R}_+^2 | n_1 \leq h_1(n_2) \text{ and } n_2 \geq s_2(\rho)\}$$

while

$$s_1(\rho) = 1 - s_2(\rho) = \min \left\{ \frac{s_1 - \rho s_2}{1 - \rho}, 1 \right\}$$

and $h_j(n_k)$ is defined implicitly by the equation

$$1 = \frac{s_j}{h_j(n_k) + \rho n_k} + \frac{s_k}{h_j(n_k) + n_k/\rho}$$
Rewriting the equation above gives us a quadratic equation in terms of $h_j(n_k)$

Since we know $h_j(n_k) > 0$ then we can just solve the quadratic equation and return the positive root

This gives us

$$h_j(n_k)^2 + \left( (\rho + \frac{1}{\rho})n_k - s_j - s_k \right) h_j(n_k) + \left( n_k^2 - \frac{s_jn_k}{\rho} - s_kn_k\rho \right) = 0$$

### 4.11.4 Simulation

Let's try simulating some of these trajectories.

We will focus in particular on whether or not innovation cycles synchronize across the two countries.

As we will see, this depends on initial conditions.

For some parameterizations, synchronization will occur for most initial conditions, while for others synchronization will be rare.

Here's the main body of code.

```julia
# Author: Shunsuke Hori

using PyPlot

function h_j(j::Integer, nk::Real, s1::Real, s2::Real, θ::Real, δ::Real, ρ::Real)
    # Find out who's h we are evaluating
    if j == 1
        sj = s1
        sk = s2
    else
        sj = s2
        sk = s1
    end

    # Coefficients on the quadratic a x^2 + b x + c = 0
    a = 1.0
    b = ((ρ + 1 / ρ) * nk - s_j - s_k)
    c = (n_k * n_k - (s_j * n_k) / ρ - sk * ρ * n_k)

    # Positive solution of quadratic form
    root = (-b + sqrt(b * b - 4 * a * c)) / (2 * a)
```

### 4.11. Globalization and Cycles
```julia
return root
end

""
Determine whether (n1, n2) is in the set DLL
""
DLL(n1::Real, n2::Real,
    s1_ρ::Real, s2_ρ::Real,
    s1::Real, s2::Real,
    θ::Real, δ::Real, ρ::Real) =
    (n1 <= s1_ρ) && (n2 <= s2_ρ)

""
Determine whether (n1, n2) is in the set DHH
""
DHH(n1::Real, n2::Real,
    s1_ρ::Real, s2_ρ::Real,
    s1::Real, s2::Real,
    θ::Real, δ::Real, ρ::Real) =
    (n1 >= h_j(1, n2, s1, s2, θ, δ, ρ)) && (n2 >= h_j(2, n1, s1, s2, θ, δ, ρ)
-> ρ))

""
Determine whether (n1, n2) is in the set DHL
""
DHL(n1::Real, n2::Real,
    s1_ρ::Real, s2_ρ::Real,
    s1::Real, s2::Real,
    θ::Real, δ::Real, ρ::Real) =
    (n1 <= s1_ρ) && (n2 <= h_j(2, n1, s1, s2, θ, δ, ρ))

""
Determine whether (n1, n2) is in the set DLH
""
DLH(n1::Real, n2::Real,
    s1_ρ::Real, s2_ρ::Real,
    s1::Real, s2::Real,
    θ::Real, δ::Real, ρ::Real) =
    (n1 >= s1_ρ) && (n2 >= s2_ρ)

""
Take a current value for (n_{(1, t)}, n_{(2, t)}) and returns the values (n_{(1, t+1)}, n_{(2, t+1)}) according to the law of motion.
""
function one_step(n1::Real, n2::Real,
    s1_ρ::Real, s2_ρ::Real,
    s1::Real, s2::Real,
    θ::Real, δ::Real, ρ::Real)
    # Depending on where we are, evaluate the right branch
    if DLL(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ)
        n1_tp1 = δ * (θ * s1_ρ + (1 - θ) * n1)
        n2_tp1 = δ * (θ * s2_ρ + (1 - θ) * n2)
    return root
end
```
```julia
elseif DHH(n1, n2, s1_, s2_, s1, s2, \theta, \delta, \rho)
    n1_tp1 = \delta \ast n1
    n2_tp1 = \delta \ast n2
elseif DHL(n1, n2, s1_, s2_, s1, s2, \theta, \delta, \rho)
    n1_tp1 = \delta \ast (\theta \ast h_j(2, n1, s1, s2, \theta, \delta, \rho) + (1 - \theta) \ast n2)
    n2_tp1 = \delta \ast (\theta \ast h_j(1, n2, s1, s2, \theta, \delta, \rho) + (1 - \theta) \ast n1)
end
return n1_tp1, n2_tp1
end

""
Given an initial condition, continues to yield new values of `n1` and `n2` ""
new_n1n2(n1_0::Real, n2_0::Real,
    s1_::Real, s2_::Real,
    s1::Real, s2::Real,
    \theta::Real, \delta::Real, \rho::Real) =
    one_step(n1_0, n2_0, s1_, s2_, s1, s2, \theta, \delta, \rho)

""
Takes initial values and iterates forward to see whether the histories eventually end up in sync.

If countries are symmetric then as soon as the two countries have the same measure of firms then they will by synchronized -- However, if they are not symmetric then it is possible they have the same measure of firms but are not yet synchronized. To address this, we check whether firms stay synchronized for `npers` periods with Euclidean norm

### Parameters
-----------
- `n1_0` : `Real`,
  Initial normalized measure of firms in country one
- `n2_0` : `Real`,
  Initial normalized measure of firms in country two
- `maxiter` : `Integer`,
  Maximum number of periods to simulate
- `npers` : `Integer`,
  Number of periods we would like the countries to have the same measure for

### Returns
-----------
- `synchronized` : `Bool`,
  Did they two economies end up synchronized
- `pers_2_sync` : `Integer`,
  The number of periods required until they synchronized

function pers_till_sync(n1_0::Real, n2_0::Real,
    s1_::Real, s2_::Real,
    n1_0::Real, n2_0::Real,
    s1_::Real, s2_::Real,
    maxiter::Integer, npers::Integer,
    \delta::Real, \theta::Real, \rho::Real,
    synchronized::Bool, pers_2_sync::Integer
)
```

4.11. Globalization and Cycles
s1::Real, s2::Real,
θ::Real, δ::Real, ρ::Real,
maxiter::Integer, npers::Integer)

# Initialize the status of synchronization
synchronized = false
pers_2_sync = maxiter
iters = 0

nsync = 0

while (~synchronized) && (iters < maxiter)
    # Increment the number of iterations and get next values
    iters += 1

    n1_t, n2_t = new_n1n2(n1_0, n2_0, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ)

    # Check whether same in this period
    if abs(n1_t - n2_t) < 1e-8
        nsync += 1
    # If not, then reset the nsync counter
    else
        nsync = 0
    end

    # If we have been in sync for npers then stop and countries
    # became synchronized nsync periods ago
    if nsync > npers
        synchronized = true
        pers_2_sync = iters - nsync
    end

    n1_0, n2_0 = n1_t, n2_t
end

return synchronized, pers_2_sync
end

function create_attraction_basis{TR <: Real}(s1_ρ::TR, s2_ρ::TR,
s1::TR, s2::TR, θ::TR, δ::TR, ρ::TR,
maxiter::Integer, npers::Integer,
->npts::Integer)

    # Create unit range with npts
    synchronized, pers_2_sync = false, 0
    unit_range = linspace(0.0, 1.0, npts)

    # Allocate space to store time to sync
    time_2_sync = Matrix{TR}(npts, npts)

    # Iterate over initial conditions
    for (i, n1_0) in enumerate(unit_range)
        for (j, n2_0) in enumerate(unit_range)
            synchronized, pers_2_sync = pers_till_sync(n1_0, n2_0, s1_ρ, s2_ρ,
            s1, s2, θ, δ, ρ,
            maxiter, npers)

            time_2_sync[i, j] = pers_2_sync
        end
    end

end

Chapter 4. Multiple Agent Models
end
end

return time_2_sync
end

# == Now we define a type for the model ==#

""

The paper "Globalization and Synchronization of Innovation Cycles" presents a two country model
with endogenous innovation cycles. Combines elements from Deneckere Judd (1985) and Helpman Krugman (1985) to allow for a
model with trade that has firms who can introduce new varieties into the economy.

We focus on being able to determine whether two countries eventually synchronize their innovation cycles. To do this, we only need a few
of the many parameters. In particular, we need the parameters listed below

### Parameters

---

- `s1` : `Real`,
  Amount of total labor in country 1 relative to total worldwide labor
- `θ` : `Real`,
  A measure of how much more of the competitive variety is used in production of final goods
- `δ` : `Real`,
  Percentage of firms that are not exogenously destroyed every period
- `ρ` : `Real`,
  Measure of how expensive it is to trade between countries

""

struct MSGSync{TR <: Real}
    s1::TR
    s2::TR
    s1_ρ::TR
    s2_ρ::TR
    θ::TR
    δ::TR
    ρ::TR
end

function MSGSync(s1::Real=0.5, θ::Real=2.5,
                  δ::Real=0.7, ρ::Real=0.2)
    # Store other cutoffs and parameters we use
    s2 = 1 - s1
    s1_ρ = min((s1 - ρ + s2) / (1 - ρ), 1)
    s2_ρ = 1 - s1_ρ

    model=MSGSync(s1, s2, s1_ρ, s2_ρ, θ, δ, ρ)
    return model
```julia
end

unpack_params(model::MSGSync) =
    model.s1, model.s2, model.θ, model.δ, model.ρ, model.s1_ρ, model.s2_ρ

""
Simulates the values of `n1` and `n2` for `T` periods

##### Parameters
-------
- `n1_0` : `Real`, Initial normalized measure of firms in country one
- `n2_0` : `Real`, Initial normalized measure of firms in country two
- `T` : `Integer`, Number of periods to simulate

##### Returns
-------
- `n1` : `Vector{TR}(ndim=1)` where TR <: Real`,
  A history of normalized measures of firms in country one
- `n2` : `Vector{TR}(ndim=1)` where TR <: Real`,
  A history of normalized measures of firms in country two
""

function simulate_n(TR <: Real)(model::MSGSync, n1_0::TR, n2_0::TR, T::Integer)
    # Unpack parameters
    s1, s2, θ, δ, ρ, s1_ρ, s2_ρ = unpack_params(model)

    # Allocate space
    n1 = Vector{TR}(T)
    n2 = Vector{TR}(T)

    # Simulate for T periods
    for t in 1:T
        # Get next values
        n1[t], n2[t] = n1_0, n2_0
        n1_0, n2_0 = new_nin2(n1_0, n2_0, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ)
    end

    return n1, n2
end

""
Takes initial values and iterates forward to see whether the histories eventually end up in sync.

If countries are symmetric then as soon as the two countries have the same measure of firms then they will by synchronized -- However, if they are not symmetric then it is possible they have the same measure of firms but are not yet synchronized. To address this, we check whether firms stay synchronized for `npers` periods with Euclidean norm

##### Parameters
-------
- `n1_0` : `Real`,
```
Initial normalized measure of firms in country one
- `n2_0` : `Real`,
Initial normalized measure of firms in country two
- `maxiter` : `Integer`,
Maximum number of periods to simulate
- `npers` : `Integer`,
Number of periods we would like the countries to have the same measure for

### Returns
------
- `synchronized` : `Bool`,
Did they two economies end up synchronized
- `pers_2_sync` : `Integer`,
The number of periods required until they synchronized

```
function pers_till_sync(model::MSGSync, n1_0::Real, n2_0::Real, 
                          maxiter::Integer=500, npers::Integer=3)
    # Unpack parameters
    s1, s2, θ, δ, ρ, s1_ρ, s2_ρ = unpack_params(model)
    return pers_till_sync(n1_0, n2_0, s1_ρ, s2_ρ, s1, s2, 
                           θ, δ, ρ, maxiter, npers)
end

function create_attraction_basis(model::MSGSync; 
                                 maxiter::Integer=250, 
                                 npers::Integer=3, 
                                 npts::Integer=50)
    # Unpack parameters
    s1, s2, θ, δ, ρ, s1_ρ, s2_ρ = unpack_params(model)
    ab = create_attraction_basis(s1_ρ, s2_ρ, s1, s2, θ, δ, 
                                 ρ, maxiter, npers, npts)
    return ab
end
```

### Time Series of Firm Measures

We write a short function below that exploits the preceding code and plots two time series

Each time series gives the dynamics for the two countries

The time series share parameters but differ in their initial condition

Heres the function

```
function plot_timeseries(n1_0::Real, n2_0::Real, 
                         s1::Real=0.5, θ::Real=2.5, 
                         δ::Real=0.5, ρ::Real=0.5, 
                         maxiter::Integer=500, npers::Integer=50)
```
\( \delta \text{k} \text{Real}=0.7, \ \rho \text{k} \text{Real}=0.2; \)

ax::PyCall.PyObject=subplots()[2]

```

Plot a single time series with initial conditions
```

# Create the MSG Model and simulate with initial conditions
model = MSGSync(s1, \( \theta \), \( \delta \), \( \rho \))
n1, n2 = simulate_n(model, n1_0, n2_0, 25)

ax[:,plot](0:24, n1, label=L"n_1\$", lw=2)
ax[:,plot](0:24, n2, label=L"n_2\$", lw=2)

ax[:,legend]()
ax[:,set_ylim](0.15, 0.8)

return ax

end

# Create figure
fig, ax = subplots(2, 1, figsize=(10, 8))

plot_timeseries(0.15, 0.35, ax=ax[1])
plot_timeseries(0.4, 0.3, ax=ax[2])

ax[1][:set_title]("Not Synchronized")
ax[2][:set_title]("Synchronized")
tight_layout()

show()

Lets see what we get
In the first case, innovation in the two countries does not synchronize.

In the second case different initial conditions are chosen, and the cycles become synchronized.

**Basin of Attraction**

Next let's study the initial conditions that lead to synchronized cycles more systematically.

We generate time series from a large collection of different initial conditions and mark those conditions with different colors according to whether synchronization occurs or not.

The next display shows exactly this for four different parameterizations (one for each subfigure).

Dark colors indicate synchronization, while light colors indicate failure to synchronize.
As you can see, larger values of $\rho$ translate to more synchronization.

You are asked to replicate this figure in the exercises.

In the solution to the exercises, you'll also find a figure with sliders, allowing you to experiment with different parameters.

Here is one snapshot from the interactive figure.
4.11.5 Exercises

Exercise 1

Replicate the figure *shown above* by coloring initial conditions according to whether or not synchronization occurs from those conditions.

4.11.6 Solutions

Exercise 1

```julia
function plot_attraction_basis(s1::Real=0.5,
                               θ::Real=2.5,
                               δ::Real=0.7,
                               ρ::Real=0.2;
                               npts::Integer=250,
                               ax=nothing)

    if ax == nothing
        fig, ax = subplots()
    end
```

4.11. Globalization and Cycles
# Create attraction basis
unitrange = linspace(0, 1, npts)
model = MSGSync(s1, θ, δ, ρ)
ab = create_attraction_basis(model, npts=npts)
cf = ax[:, pcolormesh](unitrange, unitrange, ab, cmap="viridis")

return ab, cf

fig = figure(figsize=(14, 12))

# Left - Bottom - Width - Height
ax1 = fig[:add_axes](x=[0.05, 0.475, 0.38, 0.35], label="axes0")
ax2 = fig[:add_axes](x=[0.5, 0.475, 0.38, 0.35], label="axes1")
ax3 = fig[:add_axes](x=[0.05, 0.05, 0.38, 0.35], label="axes2")
ax4 = fig[:add_axes](x=[0.5, 0.05, 0.38, 0.35], label="axes3")

params = [[0.5, 2.5, 0.7, 0.2],
           [0.5, 2.5, 0.7, 0.4],
           [0.5, 2.5, 0.7, 0.6],
           [0.5, 2.5, 0.7, 0.8]]

ab1, cf1 = plot_attraction_basis.(params[1][1], params[1][2], params[1][3],
                                params[1][4], npts=500, ax=ax1)
ab2, cf2 = plot_attraction_basis.(params[2][1], params[2][2], params[2][3],
                                params[2][4], npts=500, ax=ax2)
ab3, cf3 = plot_attraction_basis.(params[3][1], params[3][2], params[3][3],
                                params[3][4], npts=500, ax=ax3)
ab4, cf4 = plot_attraction_basis.(params[4][1], params[4][2], params[4][3],
                                params[4][4], npts=500, ax=ax4)

cbar_ax = fig[:add_axes](x=[0.9, 0.075, 0.03, 0.725])
colorbar(cf1, cax=cbar_ax)

ax1[:set_title](L"s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.2$", fontsize=22)
ax2[:set_title](L"s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.4$", fontsize=22)
ax3[:set_title](L"s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.6$", fontsize=22)
ax4[:set_title](L"s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.8$", fontsize=22)

fig[:suptitle]("Synchronized versus Asynchronized 2-cycles",
x=0.475, y=0.915, size=22)
Exercise 2

using Interact

function interact_attraction_basis(
    ρ_min::Real, ρ_step::Real, ρ_max::Real,
    maxiter_min::Integer, maxiter_step::Integer, maxiter_max::Integer,
    npts_min::Integer, npts_step::Integer, npts_max::Integer)

    # Create the figure and axis that we will plot on
    fig, ax = subplots(figsize=(12, 10))
    @manipulate for ρ=ρ_min:ρ_step:ρ_max,
        maxiter=maxiter_min:maxiter_step:maxiter_max,
        npts=npts_min:npts_step:npts_max
        withfig(fig, clear=false) do
            ax[:cla]()

            # Create model and attraction basis
            s1, θ, δ = 0.5, 2.5, 0.75
            model = MSGSync(s1, θ, δ, ρ)
ab = create_attraction_basis(model, maxiter=maxiter, npts=npts)

    # Color map with colormesh
    unitrange = linspace(0, 1, npts)
    cf = ax[:pcolormesh](unitrange, unitrange, ab, cmap="viridis")
    cbar_ax = fig[:add_axes]([0.95, 0.15, 0.05, 0.7])
    colorbar(cf, cax=cbar_ax)
end
end

interact_attraction_basis(
    0.00, 0.05, 1.0,
    50, 50, 5000,
    25, 25, 750)
4.11. Globalization and Cycles
These lectures look at important concepts in time series that are used in economics.

5.1 Covariance Stationary Processes

Contents

- Covariance Stationary Processes
  - Overview
  - Introduction
  - Spectral Analysis
  - Implementation

5.1.1 Overview

In this lecture we study covariance stationary linear stochastic processes, a class of models routinely used to study economic and financial time series.

This class has the advantage of being

1. simple enough to be described by an elegant and comprehensive theory
2. relatively broad in terms of the kinds of dynamics it can represent

We consider these models in both the time and frequency domain.

ARMA Processes

We will focus much of our attention on linear covariance stationary models with a finite number of parameters.

In particular, we will study stationary ARMA processes, which form a cornerstone of the standard theory of time series analysis.
Every ARMA processes can be represented in linear state space form

However, ARMA have some important structure that makes it valuable to study them separately

**Spectral Analysis**

Analysis in the frequency domain is also called spectral analysis

In essence, spectral analysis provides an alternative representation of the autocovariance function of a covariance stationary process

Having a second representation of this important object

- shines light on the dynamics of the process in question
- allows for a simpler, more tractable representation in some important cases

The famous *Fourier transform* and its inverse are used to map between the two representations

**Other Reading**

For supplementary reading, see

- [LS18], chapter 2
- [Sar87], chapter 11
- John Cochrane notes on time series analysis, chapter 8
- [Shi95], chapter 6
- [CC08], all

5.1.2 Introduction

Consider a sequence of random variables \{X_t\} indexed by \(t \in \mathbb{Z}\) and taking values in \(\mathbb{R}\).

Thus, \{X_t\} begins in the infinite past and extends to the infinite future a convenient and standard assumption.

As in other fields, successful economic modeling typically assumes the existence of features that are constant over time.

If these assumptions are correct, then each new observation \(X_t, X_{t+1}, \ldots\) can provide additional information about the time-invariant features, allowing us to learn from as data arrive.

For this reason, we will focus in what follows on processes that are *stationary* or become so after a transformation (see for example this lecture and this lecture).

**Definitions**

A real-valued stochastic process \{X_t\} is called *covariance stationary* if

1. Its mean \(\mu := \mathbb{E}X_t\) does not depend on \(t\)
2. For all $k$ in $\mathbb{Z}$, the $k$-th autocovariance $\gamma(k) := \mathbb{E}(X_t - \mu)(X_{t+k} - \mu)$ is finite and depends only on $k$

The function $\gamma : \mathbb{Z} \to \mathbb{R}$ is called the autocovariance function of the process.

Throughout this lecture, we will work exclusively with zero-mean (i.e., $\mu = 0$) covariance stationary processes.

The zero-mean assumption costs nothing in terms of generality, since working with non-zero-mean processes involves no more than adding a constant.

**Example 1: White Noise**

Perhaps the simplest class of covariance stationary processes is the white noise processes.

A process $\{\epsilon_t\}$ is called a white noise process if

1. $\mathbb{E}\epsilon_t = 0$

2. $\gamma(k) = \sigma^2 1\{k = 0\}$ for some $\sigma > 0$

(Here $1\{k = 0\}$ is defined to be 1 if $k = 0$ and zero otherwise)

White noise processes play the role of building blocks for processes with more complicated dynamics.

**Example 2: General Linear Processes**

From the simple building block provided by white noise, we can construct a very flexible family of covariance stationary processes: the general linear processes

$$X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}, \quad t \in \mathbb{Z}$$

(5.1)

where

- $\{\epsilon_t\}$ is white noise
- $\{\psi_t\}$ is a square summable sequence in $\mathbb{R}$ (that is, $\sum_{t=0}^{\infty} \psi_t^2 < \infty$)

The sequence $\{\psi_t\}$ is often called a linear filter.

Equation (5.1) is said to present a moving average process or a moving average representation.

With some manipulations it is possible to confirm that the autocovariance function for (5.1) is

$$\gamma(k) = \sigma^2 \sum_{j=0}^{\infty} \psi_j \psi_{j+k}$$

(5.2)

By the Cauchy-Schwartz inequality one can show that $\gamma(k)$ satisfies equation (5.2).

Evidently, $\gamma(k)$ does not depend on $t$.

---

**5.1. Covariance Stationary Processes**
Wold Decomposition

Remarkably, the class of general linear processes goes a long way towards describing the entire class of zero-mean covariance stationary processes. In particular, Wold's decomposition theorem states that every zero-mean covariance stationary process \( \{X_t\} \) can be written as

\[
X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j} + \eta_t
\]

where

- \( \{\epsilon_t\} \) is white noise
- \( \{\psi_t\} \) is square summable
- \( \eta_t \) can be expressed as a linear function of \( X_{t-1}, X_{t-2}, \ldots \) and is perfectly predictable over arbitrarily long horizons

For intuition and further discussion, see [Sar87], p. 286

AR and MA

General linear processes are a very broad class of processes. It often pays to specialize to those for which there exists a representation having only finitely many parameters.

(Experience and theory combine to indicate that models with a relatively small number of parameters typically perform better than larger models, especially for forecasting)

One very simple example of such a model is the first-order autoregressive or AR(1) process

\[
X_t = \phi X_{t-1} + \epsilon_t \quad \text{where} \quad |\phi| < 1 \quad \text{and} \quad \{\epsilon_t\} \text{ is white noise}
\]

By direct substitution, it is easy to verify that

\[
X_t = \sum_{j=0}^{\infty} \phi^j \epsilon_{t-j}
\]

Hence \( \{X_t\} \) is a general linear process

Applying (5.2) to the previous expression for \( X_t \), we get the AR(1) autocovariance function

\[
\gamma(k) = \phi^k \frac{\sigma^2}{1 - \phi^2}, \quad k = 0, 1, \ldots
\]

The next figure plots an example of this function for \( \phi = 0.8 \) and \( \phi = -0.8 \) with \( \sigma = 1 \)

using PyPlot

num_rows, num_cols = 2, 1
fig, axes = subplots(num_rows, num_cols, figsize=(10, 8))
Another very simple process is the MA(1) process (here MA means moving average)

\[ X_t = \epsilon_t + \theta \epsilon_{t-1} \]

You will be able to verify that

\[ \gamma(0) = \sigma^2 (1 + \theta^2), \quad \gamma(1) = \sigma^2 \theta, \quad \text{and} \quad \gamma(k) = 0 \quad \forall k > 1 \]
The AR(1) can be generalized to an AR\((p)\) and likewise for the MA(1).

Putting all of this together, we get the

**ARMA Processes**

A stochastic process \(\{X_t\}\) is called an *autoregressive moving average process*, or ARMA\((p,q)\), if it can be written as

\[
X_t = \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q}
\]  
(5.5)

where \(\{\epsilon_t\}\) is white noise.

An alternative notation for ARMA processes uses the *lag operator* \(L\).

**Def.** Given arbitrary variable \(Y_t\), let \(L^k Y_t := Y_{t-k}\).

It turns out that

- lag operators facilitate succinct representations for linear stochastic processes
- algebraic manipulations that treat the lag operator as an ordinary scalar are legitimate

Using \(L\), we can rewrite (5.5) as

\[
L^0 X_t - \phi_1 L^1 X_t - \cdots - \phi_p L^p X_t = L^0 \epsilon_t + \theta_1 L^1 \epsilon_t + \cdots + \theta_q L^q \epsilon_t
\]  
(5.6)

If we let \(\phi(z)\) and \(\theta(z)\) be the polynomials

\[
\phi(z) := 1 - \phi_1 z - \cdots - \phi_p z^p \quad \text{and} \quad \theta(z) := 1 + \theta_1 z + \cdots + \theta_q z^q
\]  
(5.7)

then (5.6) becomes

\[
\phi(L) X_t = \theta(L) \epsilon_t
\]  
(5.8)

In what follows we **always assume** that the roots of the polynomial \(\phi(z)\) lie outside the unit circle in the complex plane.

This condition is sufficient to guarantee that the ARMA\((p,q)\) process is covariance stationary.

In fact it implies that the process falls within the class of general linear processes described above.

That is, given an ARMA\((p,q)\) process \(\{X_t\}\) satisfying the unit circle condition, there exists a square summable sequence \(\{\psi_t\}\) with \(X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}\) for all \(t\).

The sequence \(\{\psi_t\}\) can be obtained by a recursive procedure outlined on page 79 of [CC08].

The function \(t \mapsto \psi_t\) is often called the *impulse response function*.
5.1.3 Spectral Analysis

Autocovariance functions provide a great deal of information about covariance stationary processes. In fact, for zero-mean Gaussian processes, the autocovariance function characterizes the entire joint distribution. Even for non-Gaussian processes, it provides a significant amount of information. It turns out that there is an alternative representation of the autocovariance function of a covariance stationary process, called the spectral density. At times, the spectral density is easier to derive, easier to manipulate, and provides additional intuition.

Complex Numbers

Before discussing the spectral density, we invite you to recall the main properties of complex numbers (or skip to the next section). It can be helpful to remember that, in a formal sense, complex numbers are just points \((x, y) \in \mathbb{R}^2\) endowed with a specific notion of multiplication. When \((x, y)\) is regarded as a complex number, \(x\) is called the real part and \(y\) is called the imaginary part. The modulus or absolute value of a complex number \(z = (x, y)\) is just its Euclidean norm in \(\mathbb{R}^2\), but is usually written as \(|z|\) instead of \(\|z\|\). The product of two complex numbers \((x, y)\) and \((u, v)\) is defined to be \((xu - vy, xv + yu)\), while addition is standard pointwise vector addition. When endowed with these notions of multiplication and addition, the set of complex numbers forms a field, and addition and multiplication play well together, just as they do in \(\mathbb{R}\). The complex number \((x, y)\) is often written as \(x + iy\), where \(i\) is called the imaginary unit, and is understood to obey \(i^2 = -1\). The \(x + iy\) notation provides an easy way to remember the definition of multiplication given above, because, proceeding naively,

\[
(x + iy)(u + iv) = xu - yv + i(xv + yu)
\]

Converted back to our first notation, this becomes \((xu - vy, xv + yu)\) as promised. Complex numbers can be represented in the polar form \(re^{i\omega}\) where

\[
re^{i\omega} := r(\cos(\omega) + i \sin(\omega)) = x + iy
\]

where \(x = r \cos(\omega), y = r \sin(\omega)\), and \(\omega = \arctan(y/z)\) or \(\tan(\omega) = y/x\).

Spectral Densities

Let \(\{X_t\}\) be a covariance stationary process with autocovariance function \(\gamma\) satisfying \(\sum_k \gamma(k)^2 < \infty\).
The spectral density $f$ of $\{X_t\}$ is defined as the discrete time Fourier transform of its autocovariance function $\gamma$

$$f(\omega) := \sum_{k \in \mathbb{Z}} \gamma(k) e^{-i\omega k}, \quad \omega \in \mathbb{R}$$

(Some authors normalize the expression on the right by constants such as $1/\pi$ the convention chosen makes little difference provided you are consistent)

Using the fact that $\gamma$ is even, in the sense that $\gamma(t) = \gamma(-t)$ for all $t$, we can show that

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k)$$  \hfill (5.9)

It is not difficult to confirm that $f$ is

- real-valued
- even ($f(\omega) = f(-\omega)$), and
- $2\pi$-periodic, in the sense that $f(2\pi + \omega) = f(\omega)$ for all $\omega$

It follows that the values of $f$ on $[0, \pi]$ determine the values of $f$ on all of $\mathbb{R}$ the proof is an exercise

For this reason it is standard to plot the spectral density only on the interval $[0, \pi]$

**Example 1: White Noise**

Consider a white noise process $\{\epsilon_t\}$ with standard deviation $\sigma$

It is easy to check that in this case $f(\omega) = \sigma^2$. So $f$ is a constant function

As we will see, this can be interpreted as meaning that all frequencies are equally present

(White light has this property when frequency refers to the visible spectrum, a connection that provides the origins of the term white noise)

**Example 2: AR and MA and ARMA**

It is an exercise to show that the MA(1) process $X_t = \theta \epsilon_{t-1} + \epsilon_t$ has spectral density

$$f(\omega) = \sigma^2(1 + 2\theta \cos(\omega) + \theta^2)$$  \hfill (5.10)

With a bit more effort, it is possible to show (see, e.g., p. 261 of [Sar87]) that the spectral density of the AR(1) process $X_t = \phi X_{t-1} + \epsilon_t$ is

$$f(\omega) = \frac{\sigma^2}{1 - 2\phi \cos(\omega) + \phi^2}$$  \hfill (5.11)
More generally, it can be shown that the spectral density of the ARMA process (5.5) is

\[ f(\omega) = \left( \frac{\theta(e^{i\omega})}{\phi(e^{i\omega})} \right)^2 \sigma^2 \]  

(5.12)

where

- \( \sigma \) is the standard deviation of the white noise process \( \{\epsilon_t\} \)
- the polynomials \( \phi(\cdot) \) and \( \theta(\cdot) \) are as defined in (5.7)

The derivation of (5.12) uses the fact that convolutions become products under Fourier transformations. The proof is elegant and can be found in many places, see, for example, \([Sar87]\), chapter 11, section 4. It's a nice exercise to verify that (5.10) and (5.11) are indeed special cases of (5.12).

### Interpreting the Spectral Density

Plotting (5.11) reveals the shape of the spectral density for the AR(1) model when \( \phi \) takes the values 0.8 and -0.8 respectively.

```julia
function ar1_sd(\( \omega \))
    return 1 ./ (1 - 2 * \( \cos(\omega) \) + .^2)
end

\( \omega \_s = \text{linspace}(0, \pi, 180) \)
num_rows, num_cols = 2, 1
fig, axes = subplots(num_rows, num_cols, figsize=(10, 8))

for (i, ) in enumerate((0.8, -0.8))
    ax = axes[i]
    sd = ar1_sd(\( \omega \_s \))
    label = latexstring("spectral density, \( \phi = \)", \$\$")
    ax[:plot](\( \omega \_s \), sd, "b-", alpha=0.6, lw=2, label=label)
    ax[:legend](loc="upper center")
    ax[:set](xlabel="frequency", xlim=(0, \pi))
end
```

5.1. Covariance Stationary Processes 781
These spectral densities correspond to the autocovariance functions for the AR(1) process shown above.

Informally, we think of the spectral density as being large at those $\omega \in [0, \pi]$ at which the autocovariance function seems approximately to exhibit big damped cycles.

To see the idea, let's consider why, in the lower panel of the preceding figure, the spectral density for the case $\phi = -0.8$ is large at $\omega = \pi$.

Recall that the spectral density can be expressed as

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k) = \gamma(0) + 2 \sum_{k \geq 1} (-0.8)^k \cos(\omega k)$$

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 (5.13) is large.

These ideas are illustrated in the next figure, which has $k$ on the horizontal axis.
\begin{verbatim}

# Autocovariance when \( \rho = -0.8 \)
ax = axes[1]
ax[plot](times, y1, "o-", alpha=0.6, label=L"\gamma(k)")
ax[legend](loc="upper right")
ax[set](xlim=(0, 15), yticks=(-2, 0, 2))
ax[hlines](0, 0, 15, linestyle="--", alpha=0.5)

# Cycles at frequency \( \pi \)
ax = axes[2]
ax[plot](times, y2, "o-", alpha=0.6, label=L"\cos(\pi k)")
ax[legend](loc="upper right")
ax[set](xlim=(0, 15), yticks=(-1, 0, 1))
ax[hlines](0, 0, 15, linestyle="--", alpha=0.5)

# Product
ax = axes[3]
ax[stem](times, y3, label=L"\gamma(k) \cos(\pi k)")
ax[legend](loc="upper right")
ax[set](xlim=(0, 15), ylim=(-3, 3), yticks=(-1, 0, 1, 2, 3))
ax[hlines](0, 0, 15, linestyle="--", alpha=0.5)

\end{verbatim}
On the other hand, if we evaluate $f(\omega)$ at $\omega = \pi/3$, then the cycles are not matched, the sequence $\gamma(k) \cos(\omega k)$ contains both positive and negative terms, and hence the sum of these terms is much smaller.

```python
    = -0.8
times = 0:16
y1 = [.^k ./ (1 - .^2) for k in times]
y2 = [cos.(π + k/3) for k in times]
y3 = [a + b for (a, b) in zip(y1, y2)]

num_rows, num_cols = 3, 1
fig, axes = subplots(num_rows, num_cols, figsize=(10, 8))

# Autocovariance when = -0.8
ax = axes[1]
ax[:plot](times, y1, "o-", alpha=0.6, label="\gamma(k)")
ax[:legend](loc="upper right")
ax[:set](xlim=(0, 15), yticks=(-2, 0, 2))
ax[:hlines](0, 0, 15, linestyle="--", alpha=0.5)

# Cycles at frequency π
```
In summary, the spectral density is large at frequencies $\omega$ where the autocovariance function exhibits damped cycles

**Inverting the Transformation**

We have just seen that the spectral density is useful in the sense that it provides a frequency-based perspective on the autocovariance structure of a covariance stationary process.

Another reason that the spectral density is useful is that it can be inverted to recover the autocovariance function via the *inverse Fourier transform*.

In particular, for all $k \in \mathbb{Z}$, we have
\[
\gamma(k) = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(\omega)e^{i\omega k} \, d\omega
\]  
(5.14)

This is convenient in situations where the spectral density is easier to calculate and manipulate than the autocovariance function

(For example, the expression (5.12) for the ARMA spectral density is much easier to work with than the expression for the ARMA autocovariance)

**Mathematical Theory**

This section is loosely based on \[\text{[Sar87]},\text{~p. 249-253},\text{~and included for those who}\]

- 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

\[
f = \sum_k \alpha_k h_k \quad \text{where} \quad \alpha_k := \langle f, h_k \rangle
\]

(5.15)

where \(\langle \cdot, \cdot \rangle\) denotes the inner product in \(H\)

Thus, \(f\) can be represented to any degree of precision by linearly combining basis vectors

The scalar sequence \(\alpha = \{\alpha_k\}\) is called the *Fourier coefficients* of \(f\), and satisfies \(\sum_k |\alpha_k|^2 < \infty\)

In other words, \(\alpha\) is in \(\ell_2\), the set of square summable sequences

Consider an operator \(T\) that maps \(\alpha \in \ell_2\) into its expansion \(\sum_k \alpha_k h_k \in H\)

The Fourier coefficients of \(T\alpha\) are just \(\alpha = \{\alpha_k\}\), as you can verify by confirming that \(\langle T\alpha, h_k \rangle = \alpha_k\)

Using elementary results from Hilbert space theory, it can be shown that

- \(T\) is one-to-one if \(\alpha\) and \(\beta\) are distinct in \(\ell_2\), then so are their expansions in \(H\)
- \(T\) is onto if \(f \in H\) then its preimage in \(\ell_2\) is the sequence \(\alpha\) given by \(\alpha_k = \langle f, h_k \rangle\)
- \(T\) is a linear isometry in particular \(\langle \alpha, \beta \rangle = \langle T\alpha, T\beta \rangle\)

Summarizing these results, we say that any separable Hilbert space is isometrically isomorphic to \(\ell_2\)

In essence, this says that each separable Hilbert space we consider is just a different way of looking at the fundamental space \(\ell_2\)

With this in mind, lets 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

\[
T\gamma = \sum_{k \in \mathbb{Z}} \gamma(k) \frac{e^{i\omega k}}{\sqrt{2\pi}} = \frac{1}{\sqrt{2\pi}} f(\omega) \quad (5.16)
\]

In other words, apart from a scalar multiple, the spectral density is just an transformation of \( \gamma \in \ell_2 \) under a certain linear isometry a different way to view \( \gamma \)

In particular, it is an expansion of the autocovariance function with respect to the trigonometric basis functions in \( L_2 \)

As discussed above, the Fourier coefficients of \( T\gamma \) are given by the sequence \( \gamma \), and, in particular, \( \gamma(k) = \langle T\gamma, h_k \rangle \)

Transforming this inner product into its integral expression and using (5.16) gives (5.14), justifying our earlier expression for the inverse transform

### 5.1.4 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.jl package

The module provides functions for mapping ARMA\((p, q)\) models into their

1. impulse response function
2. simulated time series
3. autocovariance function
4. spectral density

### Application

Let’s use this code to replicate the plots on pages 68–69 of [LS18]

Here are some functions to generate the plots

---

5.1. Covariance Stationary Processes 787
using QuantEcon

# == Plot functions ==#

function plot_spectral_density(arma::ARMA, ax::PyCallPyObject)
    (w, spect) = spectral_density(arma, two_pi=false)
    ax[:plot](w, spect, lw=2, alpha=0.7)
    ax[:set](title="Spectral density", xlim=(0, π),
             xlabel="frequency", ylabel="spectrum", yscale="log")
    return ax
end

function plot_spectral_density(arma::ARMA)
    fig, ax = subplots()
    plot_spectral_density(arma::ARMA, ax=ax)
    return ax
end

function plot_autocovariance(arma::ARMA, ax::PyCallPyObject)
    acov = autocovariance(arma)
    n = length(acov)
    ax[:stem](0:(n - 1), acov)
    ax[:axhline](y=0, c="red", lw=0.5)
    ax[:set](title="Autocovariance", xlim=(-0.5, n-0.5),
             xlabel="time", ylabel="autocovariance")
    return ax
end

function plot_autocovariance(arma::ARMA)
    fig, ax = subplots()
    plot_spectral_density(arma::ARMA, ax=ax)
    return ax
end

function plot_impulse_response(arma::ARMA, ax::PyCallPyObject)
    psi = impulse_response(arma)
    n = length(psi)
    ax[:stem](0:(n - 1), psi)
    ax[:axhline](y=0, c="red", lw=0.5)
    ax[:set](title="Impulse response", xlim=(-0.5, n-0.5),
             xlabel="time", ylabel="response")
    return ax
end

function plot_impulse_response(arma::ARMA)
    fig, ax = subplots()
    plot_spectral_density(arma::ARMA, ax=ax)
    return ax
end

function plot_simulation(arma::ARMA, ax::PyCallPyObject)
    X = simulation(arma)
    n = length(X)
Now let's call these functions to generate the plots

Well use the model $X_t = 0.5X_{t-1} + \epsilon_t - 0.8\epsilon_{t-2}$

```julia
θ = 0.5;
β = [0, -0.8];
arma = ARMA(θ, β, 1.0)
quad_plot(arma)
```
Explanation

The call

```julia
arma = ARMA(\(p\), \(q\), \(\sigma\))
```

creates an instance `arma` that represents the ARMA\((p, q)\) model

\[
X_t = \phi_1 X_{t-1} + \ldots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \ldots + \theta_q \epsilon_{t-q}
\]

If \(\phi\) and \(\theta\) are arrays or sequences, then the interpretation will be

- \(\phi\) holds the vector of parameters \((\phi_1, \phi_2, \ldots, \phi_p)\)
- \(\theta\) holds the vector of parameters \((\theta_1, \theta_2, \ldots, \theta_q)\)

The parameter \(\sigma\) is always a scalar, the standard deviation of the white noise

We also permit \(\phi\) and \(\theta\) to be scalars, in which case the model will be interpreted as

\[
X_t = \phi X_{t-1} + \epsilon_t + \theta \epsilon_{t-1}
\]

The two numerical packages most useful for working with ARMA models are `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 \( A_0, A_1, \ldots, A_{n-1} \) and returns the sequence \( a_0, a_1, \ldots, a_{n-1} \) defined by

\[
a_k = \frac{1}{n} \sum_{t=0}^{n-1} A_t e^{i 2\pi t / n}
\]

Thus, if we set \( A_t = f(\omega_t) \), where \( f \) is the spectral density and \( \omega_t := 2\pi t / n \), then

\[
a_k = \frac{1}{n} \sum_{t=0}^{n-1} f(\omega_t) e^{i \omega_t k} = \frac{1}{2\pi} \sum_{t=0}^{n-1} f(\omega_t) e^{i \omega_t k}, \quad \omega_t := 2\pi t / n
\]

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 (5.14) we have now shown that, for \( n \) sufficiently large, \( a_k \approx \gamma(k) \) which is exactly what we want to compute.

5.2 Estimation of Spectra
5.2.1 Overview

In a previous lecture we covered some fundamental properties of covariance stationary linear stochastic processes.

One objective for that lecture was to introduce spectral densities - a standard and very useful technique for analyzing such processes.

In this lecture we turn to the problem of estimating spectral densities and other related quantities from data.

Estimates of the spectral density are computed using what is known as a periodogram, which in turn is computed via the famous fast Fourier transform.

Once the basic technique has been explained, we will apply it to the analysis of several key macroeconomic time series.

For supplementary reading, see [Sar87] or [CC08].

5.2.2 Periodograms

Recall that the spectral density \( f \) of a covariance stationary process with autocorrelation function \( \gamma \) can be written

\[
    f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k), \quad \omega \in \mathbb{R}
\]

Now consider the problem of estimating the spectral density of a given time series, when \( \gamma \) is unknown.

In particular, let \( X_0, \ldots, X_{n-1} \) be \( n \) consecutive observations of a single time series that is assumed to be covariance stationary.

The most common estimator of the spectral density of this process is the periodogram of \( X_0, \ldots, X_{n-1} \), which is defined as

\[
    I(\omega) := \frac{1}{n} \left| \sum_{t=0}^{n-1} X_t e^{i\omega t} \right|^2, \quad \omega \in \mathbb{R}
\]  

(5.17)

(Recall that \( |z| \) denotes the modulus of complex number \( z \)).

Alternatively, \( I(\omega) \) can be expressed as

\[
    I(\omega) = \frac{1}{n} \left\{ \left( \sum_{t=0}^{n-1} X_t \cos(\omega t) \right)^2 + \left( \sum_{t=0}^{n-1} X_t \sin(\omega t) \right)^2 \right\}
\]

It is straightforward to show that the function \( I \) is even and \( 2\pi \)-periodic (i.e., \( I(\omega) = I(-\omega) \) and \( I(\omega + 2\pi) = I(\omega) \) for all \( \omega \in \mathbb{R} \)).

From these two results, you will be able to verify that the values of \( I \) on \([0, \pi]\) determine the values of \( I \) on all of \( \mathbb{R} \).

The next section helps to explain the connection between the periodogram and the spectral density.
Interpretation

To interpret the periodogram, it is convenient to focus on its values at the Fourier frequencies

$$\omega_j := \frac{2\pi j}{n}, \quad j = 0, \ldots, n - 1$$

In what sense is $I(\omega_j)$ an estimate of $f(\omega_j)$?

The answer is straightforward, although it does involve some algebra

With a bit of effort one can show that, for any integer $j > 0$,

$$\sum_{t=0}^{n-1} e^{it\omega_j} = \sum_{t=0}^{n-1} \exp \left\{ i2\pi \frac{t}{n} \right\} = 0$$

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

$$\hat{\gamma}(k) := \frac{1}{n} \sum_{t=k}^{n-1} (X_t - \bar{X})(X_{t-k} - \bar{X}), \quad k = 0, 1, \ldots, n - 1$$

This is the sample autocovariance function, the natural plug-in estimator of the autocovariance function $\gamma$ (Plug-in estimator is an informal term for an estimator found by replacing expectations with sample means)

With this notation, we can now write

$$I(\omega_j) = \hat{\gamma}(0) + 2 \sum_{k=1}^{n-1} \hat{\gamma}(k) \cos(\omega_j k)$$

Recalling our expression for $f$ given above, we see that $I(\omega_j)$ is just a sample analog of $f(\omega_j)$

Calculation

Let’s now consider how to compute the periodogram as defined in (5.17)

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

5.2. Estimation of Spectra 793
The most common way to calculate the periodogram is via the discrete Fourier transform, which in turn is implemented through the fast Fourier transform algorithm.

In general, given a sequence \( a_0, \ldots, a_{n-1} \), the discrete Fourier transform computes the sequence

\[
A_j := \sum_{t=0}^{n-1} a_t \exp\left\{ i2\pi \frac{tj}{n} \right\}, \quad j = 0, \ldots, n-1
\]

With \( a_0, \ldots, a_{n-1} \) stored in Julia array \( a \), the function call \( \text{fft}(a) \) returns the values \( A_0, \ldots, A_{n-1} \) as a Julia array.

It follows that, when the data \( X_0, \ldots, X_{n-1} \) are stored in array \( X \), the values \( I(\omega_j) \) at the Fourier frequencies, which are given by

\[
\frac{1}{n} \left| \sum_{t=0}^{n-1} X_t \exp\left\{ i2\pi \frac{tj}{n} \right\} \right|^2, \quad j = 0, \ldots, n-1
\]

can be computed by \( \text{abs}(\text{fft}(X)).^2 / \text{length}(X) \).

Note: The Julia function \( \text{abs} \) acts elementwise, and correctly handles complex numbers (by computing their modulus, which is exactly what we need).

A function called \( \text{periodogram} \) that puts all this together can be found here.

Let's generate some data for this function using the ARMA type from \( \text{QuantEcon.jl} \) (see the \textit{lecture on linear processes} for more details).

Here's a code snippet that, once the preceding code has been run, generates data from the process

\[
X_t = 0.5X_{t-1} + \epsilon_t - 0.8\epsilon_{t-2}
\]  

(5.18)

where \( \{\epsilon_t\} \) is white noise with unit variance, and compares the periodogram to the actual spectral density.

```julia
using QuantEcon
using Plots
pyplot()

n = 40            # Data size
phi = 0.5         # AR parameter
theta=[0, -0.8]   # MA parameter
sigma=1.0
lp = ARMA(phi, theta, sigma)
X = simulation(lp, ts_length=n)

x, y = periodogram(X)
x_sd, y_sd = spectral_density(lp, two_pi=false, res=120)

plot(x, y, linecolor="blue", linewidth=2, linealpha=0.5, lab="periodogram")
plot!(x_sd, y_sd, linecolor="red", linewidth=2, linealpha=0.8, lab="spectral density")
```
Running this should produce a figure similar to this one

This estimate looks rather disappointing, but the data size is only 40, so perhaps its not surprising that the estimate is poor

However, if we try again with \( n = 1200 \) the outcome is not much better
The periodogram is far too irregular relative to the underlying spectral density.

This brings us to our next topic.

5.2.3 Smoothing

There are two related issues here:

One is that, given the way the fast Fourier transform is implemented, the number of points $\omega$ at which $I(\omega)$ is estimated increases in line with the amount of data.

In other words, although we have more data, we are also using it to estimate more values.

A second issue is that densities of all types are fundamentally hard to estimate without parametric assumptions.

Typically, nonparametric estimation of densities requires some degree of smoothing.

The standard way that smoothing is applied to periodograms is by taking local averages.
In other words, the value $I(\omega_j)$ is replaced with a weighted average of the adjacent values

$$I(\omega_{j-p}), I(\omega_{j-p+1}), \ldots, I(\omega_j), \ldots, I(\omega_{j+p})$$

This weighted average can be written as

$$I_S(\omega_j) := \sum_{\ell=-p}^{p} w(\ell) I(\omega_{j+\ell}) \quad (5.19)$$

where the weights $w(-p), \ldots, w(p)$ are a sequence of $2p + 1$ nonnegative values summing to one.

In generally, larger values of $p$ indicate more smoothing. More on this below.

The next figure shows the kind of sequence typically used.

Note the smaller weights towards the edges and larger weights in the center, so that more distant values from $I(\omega_j)$ have less weight than closer ones in the sum (5.19).

```julia
function hanning_window(M)
    w = [0.5 - 0.5 * cos(2 * pi * n / (M - 1)) for n = 0:(M-1)]
    return w
end

window = hanning_window(25) / sum(hanning_window(25))
x = linspace(-12, 12, 25)
plot(x, window, color="darkblue", title="Hanning window", ylabel="Weights",
     xlabel="Position in sequence of weights", legend=false, grid=false)
```

5.2. Estimation of Spectra
Estimation with Smoothing

Our next step is to provide code that will not only estimate the periodogram but also provide smoothing as required.

Such functions have been written in `estspec.jl` and are available once you've installed `QuantEcon.jl`.

The GitHub listing displays three functions, `smooth()`, `periodogram()`, `ar_periodogram()`. We will discuss the first two here and the third one below.

The `periodogram()` function returns a periodogram, optionally smoothed via the `smooth()` function. Regarding the `smooth()` function, since smoothing adds a nontrivial amount of computation, we have applied a fairly terse array-centric method based around `conv`.

Readers are left either to explore or simply to use this code according to their interests.

The next three figures each show smoothed and unsmoothed periodograms, as well as the population or true spectral density.

(The model is the same as before, see equation (5.18) and there are 400 observations.)

From top figure to bottom, the window length is varied from small to large.
In looking at the figure, we can see that for this model and data size, the window length chosen in the middle figure provides the best fit.

Relative to this value, the first window length provides insufficient smoothing, while the third gives too much smoothing.

5.2. Estimation of Spectra
Of course in real estimation problems the true spectral density is not visible and the choice of appropriate smoothing will have to be made based on judgement/priors or some other theory.

**Pre-Filtering and Smoothing**

In the code listing we showed three functions from the file `estspec.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 (5.19).

The underlying assumption that makes this a good idea is that the true spectral density is relatively regular, the value of \( I(\omega) \) is close to that of \( I(\omega') \) when \( \omega \) is close to \( \omega' \).

This will not be true in all cases, but it is certainly true for white noise.

For white noise, \( I \) is as regular as possible: **it is a constant function**.

In this case, values of \( I(\omega') \) at points \( \omega' \) near to \( \omega \) provided the maximum possible amount of information about the value \( I(\omega) \).

Another way to put this is that if \( I \) is relatively constant, then we can use a large amount of smoothing without introducing too much bias.

**The AR(1) Setting**

Let's examine this idea more carefully in a particular setting where the data are assumed to be generated by an AR(1) process.

(More general ARMA settings can be handled using similar techniques to those described below.)

Suppose in particular that \( \{X_t\} \) is covariance stationary and AR(1), with

\[
X_{t+1} = \mu + \phi X_t + \epsilon_{t+1} \quad (5.20)
\]
where \( \mu \) and \( \phi \in (-1, 1) \) are unknown parameters and \( \{\epsilon_t\} \) is white noise.

It follows that if we regress \( X_{t+1} \) on \( X_t \) and an intercept, the residuals will approximate white noise.

Let

- \( g \) be the spectral density of \( \{\epsilon_t\} \) a constant function, as discussed above.
- \( I_0 \) be the periodogram estimated from the residuals an estimate of \( g \).
- \( f \) be the spectral density of \( \{X_t\} \) the object we are trying to estimate.

In view of *an earlier result* we obtained while discussing ARMA processes, \( f \) and \( g \) are related by

\[
f(\omega) = \left| \frac{1}{1 - \phi e^{i\omega}} \right|^2 g(\omega) \tag{5.21}\]

This suggests that the recoloring step, which constructs an estimate \( I \) of \( f \) from \( I_0 \), should set

\[
I(\omega) = \left| \frac{1}{1 - \hat{\phi} e^{i\omega}} \right|^2 I_0(\omega)
\]

where \( \hat{\phi} \) is the OLS estimate of \( \phi \).

The code for `ar_periodogram()` the third function in `estspec.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 (5.20) with \( \mu = 0 \) and \( \phi = -0.9 \).

Each time series is of length 150.

The difference between the three subfigures is just randomness each one uses a different draw of the time series.
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
5.2.4 Exercises

Exercise 1

Replicate this figure (modulo randomness)

The model is as in equation (5.18) and there are 400 observations

For the smoothed periodogram, the window type is hamming

Exercise 2

Replicate this figure (modulo randomness)

The model is as in equation (5.20), with $\mu = 0$, $\phi = -0.9$ and 150 observations in each time series

All periodograms are fit with the hamming window and window length of 65

5.2.5 Solutions

Exercise 1

```julia
srand(42)  # reproducible results

n = 400
phi = 0.5
theta = [0, -0.8]
sigma = 1.0
lp = ARMA(phi, theta, 1.0)
X = simulation(lp, ts_length=n)
xs = []
x_sds = []
x_sms = []
ys = []
y_sds = []
y_sms = []
titles = []

for (i, wl) in enumerate([15, 55, 175])  # window lengths
    x, y = periodogram(X)
    push!(xs, x)
    push!(ys, y)
    x_sd, y_sd = spectral_density(lp, two_pi=false, res=120)
    push!(x_sds, x_sd)
    push!(y_sds, y_sd)
    x, y_smoothed = periodogram(X, "hamming", wl)
    push!(x_sms, x)
    push!(y_sms, y_smoothed)
```

5.2. Estimation of Spectra


```julia
    t = "window length = $wl"
push!(titles, t)
end

plot(xs, ys, layout=(3,1), color=:blue, alpha=0.5,
     linewidth=2, label=["periodogram" ""]
plot!(x_sds, y_sds, layout=(3,1), color=:red, alpha=0.8,
     linewidth=2, label=["spectral density" ""]
plot!(x_sms, y_sms, layout=(3,1), color=:black,
     linewidth=2, label=["smoothed periodogram" ""]
plot!(title=reshape(titles,1,length(titles)))

Exercise 2

lp2 = ARMA(-0.9, 0.0, 1.0)
w1 = 65
p=plot(layout=(3,1))

for i in 1:3
    X = simulation(lp2,ts_length=150)
    plot!(p[i],xlims=(0,pi))
    x_sd, y_sd = spectral_density(lp2,two_pi=false, res=180)
    plot!(p[i],x_sd, y_sd, linecolor=:red, linestyle=:solid,
         yscale=:log10, linewidth=2, linealpha=0.75,
```

Chapter 5. Time Series Models
5.3 Additive Functionals
5.3.1 Overview

Some time series are nonstationary

For example, output, prices, and dividends are typically nonstationary, due to irregular but persistent growth

Which kinds of models are useful for studying such time series?

Hansen and Scheinkman [HS09] analyze two classes of time series models that accommodate growth

They are:

1. **additive functionals** that display random arithmetic growth

2. **multiplicative functionals** that display random geometric growth

These two classes of processes are closely connected

For example, if a process \( \{y_t\} \) is an additive functional and \( \phi_t = \exp(y_t) \), then \( \{\phi_t\} \) is a multiplicative functional

Hansen and Sargent [HS17] (chs. 5 and 6) describe discrete time versions of additive and multiplicative functionals

In this lecture we discuss the former (i.e., additive functionals)

In the next lecture we discuss multiplicative functionals

We also consider fruitful decompositions of additive and multiplicative processes, a more in depth discussion of which can be found in Hansen and Sargent [HS17]

5.3.2 A Particular Additive Functional

This lecture focuses on a particular type of additive functional: a scalar process \( \{y_t\}_{t=0}^{\infty} \) whose increments are driven by a Gaussian vector autoregression

It is simple to construct, simulate, and analyze

This additive functional consists of two components, the first of which is a **first-order vector autoregression** (VAR)

\[
x_{t+1} = Ax_t + Bz_{t+1}
\] (5.22)

Here
• \( x_t \) is an \( n \times 1 \) vector,
• \( A \) is an \( n \times n \) stable matrix (all eigenvalues lie within the open unit circle),
• \( z_{t+1} \sim \mathcal{N}(0, I) \) is an \( m \times 1 \) i.i.d. shock,
• \( B \) is an \( n \times m \) matrix, and
• \( x_0 \sim \mathcal{N}(\mu_0, \Sigma_0) \) is a random initial condition for \( x \)

The second component is an equation that expresses increments of \( \{y_t\}_{t=0}^\infty \) as linear functions of
• a scalar constant \( \nu \),
• the vector \( x_t \), and
• the same Gaussian vector \( z_{t+1} \) that appears in the VAR (5.22)

In particular,

\[
y_{t+1} - y_t = \nu + D x_t + F z_{t+1} \tag{5.23}
\]

Here \( y_0 \sim \mathcal{N}(\mu_y, \Sigma_y) \) is a random initial condition

The nonstationary random process \( \{y_t\}_{t=0}^\infty \) displays systematic but random arithmetic growth

A linear state space representation

One way to represent the overall dynamics is to use a linear state space system

To do this, we set up state and observation vectors

\[
\begin{bmatrix} \hat{x}_t \\ \hat{y}_t \end{bmatrix} = \begin{bmatrix} 1 \\ x_t \\ y_t \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \hat{y}_t \end{bmatrix} = \begin{bmatrix} x_t \\ y_t \end{bmatrix}
\]

Now we construct the state space system

\[
\begin{bmatrix} 1 \\ x_{t+1} \\ y_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & A & 0 \\ \nu & D' & 1 \end{bmatrix} \begin{bmatrix} 1 \\ x_t \\ y_t \end{bmatrix} + \begin{bmatrix} 0 \\ B \\ F' \end{bmatrix} z_{t+1}
\]

\[
\begin{bmatrix} x_t \\ y_t \end{bmatrix} = \begin{bmatrix} 0 & I & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ x_t \\ y_t \end{bmatrix}
\]

This can be written as

\[
\begin{align*}
\hat{x}_{t+1} &= \hat{A} \hat{x}_t + \hat{B} z_{t+1} \\
\hat{y}_t &= \hat{D} \hat{x}_t
\end{align*}
\]

which is a standard linear state space system

To study it, we could map it into an instance of LSS from QuantEcon.jl

We will in fact use a different set of code for simulation, for reasons described below

5.3. Additive Functionals
5.3.3 Dynamics

Let’s run some simulations to build intuition

In doing so well assume that $z_{t+1}$ is scalar and that $\tilde{x}_t$ follows a 4th-order scalar autoregression

$$
\tilde{x}_{t+1} = \phi_1 \tilde{x}_t + \phi_2 \tilde{x}_{t-1} + \phi_3 \tilde{x}_{t-2} + \phi_4 \tilde{x}_{t-3} + \sigma z_{t+1}
$$  \hspace{1cm} (5.24)

Let the increment in $\{y_t\}$ obey

$$
y_{t+1} - y_t = \nu + \tilde{x}_t + \sigma z_{t+1}
$$

with an initial condition for $y_0$

While (5.24) is not a first order system like (5.22), we know that it can be mapped into a first order system

- for an example of such a mapping, see this example

In fact this whole model can be mapped into the additive functional system definition in (5.22) – (5.23) by appropriate selection of the matrices $A, B, D, F$

You can try writing these matrices down now as an exercise the correct expressions will appear in the code below

Simulation

When simulating we embed our variables into a bigger system

This system also constructs the components of the decompositions of $y_t$ and of $\exp(y_t)$ proposed by Hansen and Scheinkman [HS09]

All of these objects are computed using the code below

```julia
#=

Author: Shunsuke Hori

#=
using QuantEcon
using PyPlot
using Distributions

""
This type transforms an additive (multiplicative)
functional into a QuantEcon linear state space system.
""

struct AMF_LSS_VAR{TF<:AbstractFloat, TI<:Integer}
    A::Array{TF, 2}
    B::Array{TF, 2}
    D::Array{TF, 2}
    F::Array{TF, 2}
    \nu::Array{TF, 2}
end
```

Chapter 5. Time Series Models
nx::TI
nk::TI
nm::TI
iss::LSS
end

function AMF_LSS_VAR(A::Array, B::Array,
D::Union{RowVector, Array},
F::Union{Void, Array} = nothing;
ν::Union{Void, Array} = nothing)

if typeof(B) <: Vector
    B = reshape(B, length(B), 1)
end

# Unpack required elements
nx, nk = size(B)

# checking the dimension of D (extended from the scalar case)
if ndims(D) > 1
    nm = size(D, 1)
    if typeof(D) <: RowVector
        D = convert(Matrix, D)
    end
else
    nm = 1
    D = reshape(D, 1, length(D))
end

# Set F
if F == nothing
    F = zeros(nk, 1)
elseif ndims(F) == 1
    F = reshape(F, length(F), 1)
end

# Set ν
if ν == nothing
    ν = zeros(nm, 1)
elseif ndims(ν) == 1
    ν = reshape(ν, length(ν), 1)
else
    throw(ArgumentError("ν must be column vector!"))
end

if size(ν, 1) != size(D, 1)
    error("The size of ν is inconsistent with D!"
end

# Construct BIG state space representation
iss = construct_ss(A, B, D, F, ν, nx, nk, nm)

return AMF_LSS_VAR(A, B, D, F, ν, nx, nk, nm, iss)
end
function construct_ss(A::Array, B::Array, D::Union{RowVector, Array}, F::Real, ν::Real) where TI <: Integer
    H, g = additive_decomp(A, B, D, F, nx)

    # Auxiliary blocks with 0's and 1's to fill out the lss matrices
    nx0c = zeros(nx, 1)
    nx0r = zeros(1, nx)
    nx1 = ones(1, nx)
    nk0 = zeros(1, nk)
    ny0c = zeros(nm, 1)
    ny0r = zeros(1, nm)
    ny1m = eye(nm)
    ny0m = zeros(nm, nm)
    nx0m = zeros(D)

    # Build A matrix for LSS
    # Order of states is: [1, t, xt, yt, mt]
    A1 = hcat(1, 0, nx0r, ny0r, ny0r)     # Transition for 1
    A2 = hcat(1, 1, nx0r, ny0r, ny0r)     # Transition for t
    A3 = hcat(nx0c, nx0c, A, ny0x0m', ny0x0m')     # Transition for x_{t+1}
    A4 = hcat(ν, ny0c, D, ny1m, ny0m)     # Transition for y_{t+1}
    A5 = hcat(ny0c, ny0c, nyx0m, ny0m, ny1m)     # Transition for m_{t+1}
    Abar = vcat(A1, A2, A3, A4, A5)

    # Build B matrix for LSS
    Bbar = vcat(nk0, nk0, B, F, H)

    # Build G matrix for LSS
    # Order of observation is: [xt, yt, mt, st, tt]
    G1 = hcat(nx0c, nx0c, eye(nx), ny0x0m', ny0x0m')     # Selector for x_{t}
    G2 = hcat(ny0c, ny0c, nyx0m, ny0x0m, ny0m)     # Selector for y_{t}
    G3 = hcat(ny0c, ny0c, nyx0m, ny0m, ny1m)     # Selector for st
    G4 = hcat(ny0c, ny0c, -g, ny0m, ny0m)     # Selector for tt
    G5 = hcat(ny0c, ν, nx0m, ny0m, ny0m)     # Selector for trend
    Gbar = vcat(G1, G2, G3, G4, G5)

    # Build LSS type
    x0 = hcat(1, 0, nx0r, ny0r, ny0r)
5.3. Additive Functionals

```julia
SO = zeros(length(x0), length(x0))
iss = LSS(Abar, Bbar, Gbar, zeros(nx+4nm, 1), x0, SO)

    return iss
end

""
Return values for the martingale decomposition
- \( \nu \): unconditional mean difference in \( Y \)
- \( H \): coefficient for the (linear) martingale component \( (kappa_a) \)
- \( g \): coefficient for the stationary component \( g(x) \)
- \( Y_0 \): it should be the function of \( X_0 \) (for now set it to 0.0)
""
function additive_decomp(A::Array, B::Array, D::Array, F::Union{Array, Real}, nx::Integer)
    I = eye(nx)
    A_res = \( I-A, I \)
    g = D * A_res
    H = F + D * A_res + B

    return H, g
end

""
Return values for the multiplicative decomposition (Example 5.4.4.)
- \( \nu_{\text{tilde}} \): eigenvalue
- \( H \): vector for the Jensen term
""
function multiplicative_decomp(A::Array, B::Array, D::Array, F::Union{Array, Real}, nx::Integer)
    \( H, g = additive_decomp(A, B, D, F, nx) \)
    \( \nu_{\text{tilde}} = \nu + 0.5*\text{diag}(H*H') \)

    return H, g, \( \nu_{\text{tilde}} \)
end

function loglikelihood_path(amf::AMF_LSS_VAR, x::Array, y::Array)
    k, T = size(y)
    FF = F*F'
    FFinv = inv(FF)
    temp = y[:, 2:end]-y[:, 1:end-1] - D*x[:, 1:end-1]
    obs = temp .* FFinv .* temp
    obssum = cumsum(obs)
    scalar = (log(det(FF)) + k*log(2*pi))\cdot collect(1:T)

    return -0.5\cdot(obssum + scalar)
end

function loglikelihood(amf::AMF_LSS_VAR, x::Array, y::Array)
    llh = loglikelihood_path(amf, x, y)
```

811
return llh[end]
end

""
Plots for the additive decomposition
"

function plot_additive(amf::AMF_LSS_VAR, T::Integer;
    npaths::Integer=25, show_trend::Bool=true)

    # Pull out right sizes so we know how to increment
    nx, nk, nm = amf.nx, amf.nk, amf.nm

    # Allocate space (nm is the number of additive functionals - we want
    # npaths for each)
    mpath = Array{Real}(nm*npaths, T)
    mbounds = Array{Real}(nm*2, T)
    spath = Array{Real}(nm*npaths, T)
    sbounds = Array{Real}(nm*2, T)
    tpath = Array{Real}(nm*npaths, T)
    ypath = Array{Real}(nm*npaths, T)

    # Simulate for as long as we wanted
    moment_generator = moment_sequence(amf.lss)
    state = start(moment_generator)

    # Pull out population moments
    for t in 1:T
        tmoms, state = next(moment_generator, state)
        ymeans = tmoms[2]
        yvar = tmoms[4]

        # Lower and upper bounds - for each additive functional
        for ii in 1:nm
            li, ui = (ii-1)*2+1, ii*2
            if sqrt(yvar[nx+nm+ii, nx+nm+ii]) != 0.0
                madd_dist = Normal(ymeans[nx+nm+ii], sqrt(yvar[nx+nm+ii, nx+nm+ii]))
                mbounds[li, t] = quantile(madd_dist, 0.01)
                mbounds[ui, t] = quantile(madd_dist, 0.99)
            elseif sqrt(yvar[nx+nm+ii, nx+nm+ii]) == 0.0
                mbounds[li, t] = ymeans[nx+nm+ii]
                mbounds[ui, t] = ymeans[nx+nm+ii]
            else
                error("standard error is negative")
            end

            if sqrt(yvar[nx+2*nm+ii, nx+2*nm+ii]) != 0.0
                sadd_dist = Normal(ymeans[nx+2*nm+ii], sqrt(yvar[nx+2*nm+ii, nx+2*nm+ii]))
                sbounds[li, t] = quantile(sadd_dist, 0.01)
                sbounds[ui, t] = quantile(sadd_dist, 0.99)
            elseif sqrt(yvar[nx+2*nm+ii, nx+2*nm+ii]) == 0.0
                sbounds[li, t] = ymeans[nx+2*nm+ii]
            else
                error("standard error is negative")
            end
        end
    end

Chapter 5. Time Series Models
sbounds[ui, t] = ymeans[nx+2*nm+ii]
else
    error("standard error is negative")
end

end
end

# Pull out paths
for n in 1:npaths
    x, y = simulate(amf.lss,T)
    for ii in 0:nm-1
        ypath[npaths*ii+n, :] = y[nx+ii+1, :]
        mpath[npaths*ii+n, :] = y[nx+nm + ii+1, :]
        spath[npaths*ii+n, :] = y[nx+2*nm + ii+1, :]
        tpath[npaths*ii+n, :] = y[nx+3*nm + ii+1, :]
    end
end

add_figs = Array{Any}(nm)

for ii in 0:nm-1
    li, ui = npaths*(ii), npaths*(ii+1)
    LI, UI = 2*(ii), 2*(ii+1)
    add_figs[ii+1] =
        plot_given_paths(T, ypath[li+1:ui, :], mpath[li+1:ui, :],
                        spath[li+1:ui, :],
                        tpath[li+1:ui, :], mbounds[LI+1:UI, :],
                        sbounds[LI+1:UI, :],
                        show_trend=show_trend)
        add_figs[ii+1][:suptitle]( L"Additive decomposition of $y_{$(ii+1)$}$",
                        fontsize=14 )
    end

return add_figs
end

""
Plots for the multiplicative decomposition
""

function plot_multiplicative(amf::AMF_LSS_VAR, T::Integer,
                               npaths::Integer=25, show_trend::Bool=true)
    # Pull out right sizes so we know how to increment
    nx, nk, nm = amf.nx, amf.nk, amf.nm
    # Matrices for the multiplicative decomposition
    H, g, h_tilde = multiplicative_decomp(A, B, D, F, ν, nx)

    # Allocate space (nm is the number of functionals - we want npaths for
    # each)
    mpath_mult = Array{Real}(nm*npaths, T)
    mbounds_mult = Array{Real}(nm*2, T)
    spath_mult = Array{Real}(nm*npaths, T)

5.3. Additive Functionals 813
sbounds_mult = Array{Real}(nm+2, T)
tpath_mult = Array{Real}(nm+npaths, T)
ypath_mult = Array{Real}(nm+npaths, T)

# Simulate for as long as we wanted
moment_generator = moment_sequence(amf.lss)
state = start(moment_generator)

# Pull out population moments
for t in 1:T
    tmeans, state = next(moment_generator, state)
    yvar = tmeans[2]
end

# Lower and upper bounds - for each multiplicative functional
for ii in 1:nm
    li, ui = (ii-1)+2+1, ii+2
    if yvar[nx+nm+ii, nx+nm+ii] != 0
        Mdist = LogNormal(ymeans[nx+nm+ii]- t*0.5*diag(H * H')[ii],
            sqrt(yvar[nx+nm+ii, nx+nm+ii]))
        mbounds_mult[li, t] = quantile(Mdist, 0.01)
        mbounds_mult[ui, t] = quantile(Mdist, 0.99)
    elseif yvar[nx+nm+ii, nx+nm+ii] == 0
        mbounds_mult[li, t] = exp.(ymeans[nx+nm+ii]- t*0.5*diag(H * H 
            '))[ii]
        mbounds_mult[ui, t] = exp.(ymeans[nx+nm+ii]- t*0.5*diag(H * H 
            '))[ii]
    else
        error("standard error is negative")
    end
    if yvar[nx+2+nm+ii, nx+2+nm+ii] != 0
        Sdist = LogNormal(-ymeans[nx+2+nm+ii],
        sqrt(yvar[nx+2+nm+ii, nx+2+nm+ii]))
        sbounds_mult[li, t] = quantile(Sdist, 0.01)
        sbounds_mult[ui, t] = quantile(Sdist, 0.99)
    elseif yvar[nx+2+nm+ii, nx+2+nm+ii] == 0
        sbounds_mult[li, t] = exp.(ymeans[nx+2+nm+ii])
        sbounds_mult[ui, t] = exp.(ymeans[nx+2+nm+ii])
    else
        error("standard error is negative")
    end
end

# Pull out paths
for n in 1:npaths
    x, y = simulate(amf.lss,T)
    for ii in 0:nm-1
        ypath_mult[npaths*ii+n, :) = exp.(y[nx+ii+1, :])
        tpath_mult[npaths*ii+n, :) =
            exp.(y[nx+nm + ii+1, :) - collect(1:T)*0.5*diag(H * H')[ii+1])
        spath_mult[npaths*ii+n, :) = 1./exp.(-y[nx+2*nm + ii+1, :])
        tpath_mult[npaths*ii+n, :) =
            exp.(y[nx+3*nm + ii+1, :) + collect(1:T)*0.5*diag(H * H 
                '))[ii+1])
end
end

mult_figs = Array{Any}(nm)

for ii in 0:nm-1
    li, ui = npaths*(ii), npaths*(ii+1)
    LI, UI = 2*(ii), 2*(ii+1)
    mult_figs[ii+1] =
        plot_given_paths(T, ypath_mult[li+1:ui, :], mpath_mult[li+1:ui, :],
                       spath_mult[li+1:ui, :], tpath_mult[li+1:ui, :],
                       mbounds_mult[LI+1:UI, :], sbounds_mult[LI+1:UI, :],
                       horline = 1.0, show_trend=show_trend)
        mult_figs[ii+1][:suptitle]( L"Multiplicative decomposition of $y_{$
          @(ii+1)$}"$",
                       fontsize=14)
    end
return mult_figs
end

function plot_martingales(amf::AMF_LSS_VAR, T::Integer, npaths::Integer=25)
    # Pull out right sizes so we know how to increment
    nx, nk, nm = amf.nx, amf.nk, amf.nm
    # Matrices for the multiplicative decomposition
    # Allocate space (nm is the number of functionals - we want npaths for each)
    mpath_mult = Array{Real}(nm*npaths, T)
    mbounds_mult = Array{Real}(nm*2, T)

    # Simulate for as long as we wanted
    moment_generator = moment_sequence(amf.lss)
    state = start(moment_generator)
    # Pull out population moments
    for t in 1:T
        tmoms, state = next(moment_generator, state)
        ymeans = tmoms[2]
        yvar = tmoms[4]

        # Lower and upper bounds - for each functional
        for ii in 1:nm
            li, ui = (ii-1)*2+1, ii*2
            if yvar[nx+nm+ii, nx+nm+ii] != 0.0
                Mdist = LogNormal(ymeans[nx+nm+ii]-t*(.5)*diag(H*H')*[ii],
                                   sqrt(yvar[nx+nm+ii, nx+nm+ii]))
                mbounds_mult[li, t] = quantile(Mdist, 0.01)
                mbounds_mult[ui, t] = quantile(Mdist, 0.99)
```julia
elseif yvar[nx+nm+ii, nx+nm+ii] == 0.0
    mbounds_mult[li, t] = ymeans[nx+nm+ii] - t * .5 * diag(H*H')[ii]
    mbounds_mult[ui, t] = ymeans[nx+nm+ii] - t * .5 * diag(H*H')[ii]
else
    error("standard error is negative")
end
end

# Pull out paths
for n in 1:npaths
    x, y = simulate(amf.lss, T)
    for ii in 0:nm-1
        mpath_mult[npaths*ii+n, :] = exp.(y[nx+nm + ii+1, :] - (1:T)*0.5*diag(H*H')[ii+1])
    end
end

mart_figs = Array{Any}(nm)

for ii in 0:nm-1
    li, ui = npaths*(ii), npaths*(ii+1)
    LI, UI = 2*(ii), 2*(ii+1)
    mart_figs[ii+1] = plot_martingale_paths(T, mpath_mult[li+1:ui, :], mbounds_mult[LI+1:UI, :],
    →horline=1)
    mart_figs[ii+1][:supitle]("Martingale components for many paths of
    →$y_{\text{ii+1}}$",
    →fontsize=14)
end

return mart_figs
end

function plot_given_paths(T::Integer,
    ypath::Array, mpath::Array, spath::Array,
    →tpath::Array,
    mbounds::Array, sbounds::Array; horline::Real=0.0,
    show_trend::Bool = true)

    # Allocate space
    trange = 1:T

    # Create figure
    fig, ax = subplots(2, 2, sharey=true, figsize=(15, 8))

    # Plot all paths together
    ax[1, 1][:plot](trange, ypath[1, :], label="$y_t$", color="k")
    ax[1, 1][:plot](trange, mpath[1, :], label="$m_t$", color="m")
    ax[1, 1][:plot](trange, spath[1, :], label="$s_t$", color="g")
    if show_trend == true
        ax[1, 1][:plot](trange, tpath[1, :], label="t-t", color="r")
    end
```
For now, we just plot $y_t$ and $x_t$, postponing until later a description of exactly how we compute them.

### 5.3. Additive Functionals

```julia
ax[1, 1][:axhline](horline, color="k", linestyle = "-.")
ax[1, 1][:set_title]("One Path of All Variables")
ax[1, 1][:legend](loc="top left")

# Plot Martingale Component
ax[1, 2][:plot](trange, mpath[1, :], "m")
ax[1, 2][:plot](trange, mpath', alpha=0.45, color="m")
ub = mbounds[2, :]
lb = mbounds[1, :]
ax[1, 2][:fill_between](trange, lb, ub, alpha=0.25, color="m")
ax[1, 2][:set_title]("Martingale Components for Many Paths")
ax[1, 2][:axhline](horline, color="k", linestyle = "-.")

# Plot Stationary Component
ax[2, 1][:plot](spath[1, :], color="g")
ax[2, 1][:plot](spath', alpha=0.25, color="g")
ub = sbounds[2, :]
lb = sbounds[1, :]
ax[2, 1][:fill_between](trange, lb, ub, alpha=0.25, color="g")
ax[2, 1][:axhline](horline, color="k", linestyle = "-.")
ax[2, 1][:set_title]("Stationary Components for Many Paths")

# Plot Trend Component
if show_trend == true
  ax[2, 2][:plot](tpath', color="r")
end
ax[2, 2][:set_title]("Trend Components for Many Paths")
ax[2, 2][:axhline](horline, color="k", linestyle = "-.")

return fig
end

function plot_martingale_paths(T::Integer,
  mpath::Array, mbounds::Array;
  horline::Real=1,
  show_trend::Bool = false)  
  
  # Allocate space
  trange = 1:T

  # Create figure
  fig, ax = subplots(1, 1, figsize=(10, 6))

  # Plot Martingale Component
  ub = mbounds[2, :]
lb = mbounds[1, :]
ax[:fill_between](trange, lb, ub, color="#ffccff")
ax[:axhline](horline, color="k", linestyle = "-.")
ax[:plot](trange, mpath', linewidth=0.25, color="#4c4c4c")

  return fig
end
```
_1, _2, _3, _4 = 0.5, -0.2, 0, 0.5
σ = 0.01
ν = 0.01 # Growth rate

# A matrix should be n x n
A = [ _1 _2 _3 _4;
      1 0 0 0;
      0 1 0 0;
      0 0 1 0]

# B matrix should be n x k
B = [σ, 0, 0, 0]

D = [1 0 0 0] * A
F = dot([1, 0, 0, 0], vec(B))

amf = AMF_LSS_VAR(A, B, D, F, ν)

T = 150
x, y = simulate(amf.lss, T)

fig, ax = subplots(2, 1, figsize = (10, 9))

ax[1][:plot](1:T, y[amf.ny+1, :], color="k")
ax[1][:set_title]("A particular path of \(\mathbf{y}_t\)")
ax[2][:plot](1:T, y[1, :], color="g")
ax[2][:axhline](0, color="k", linestyle="--")
ax[2][:set_title]("Associated path of \(\mathbf{x}_t\)")
Notice the irregular but persistent growth in $y_t$

**Decomposition**

Hansen and Sargent \([HS17]\) describe how to construct a decomposition of an additive functional into four parts:

- a constant inherited from initial values $x_0$ and $y_0$
- a linear trend
- a martingale
- an (asymptotically) stationary component

To attain this decomposition for the particular class of additive functionals defined by (5.22) and (5.23), we
first construct the matrices

\[
H := F + B'(I - A')^{-1} D \\
g := D'(I - A)^{-1}
\]

Then the Hansen-Scheinkman \cite{HS09} decomposition is

\[
y_t = \underbrace{t \nu}_{\text{trend component}} + \sum_{j=1}^t H z_j - \underbrace{gx_t}_{\text{stationary component}} + \underbrace{gx_0 + y_0}_{\text{initial conditions}}
\]

At this stage you should pause and verify that \(y_{t+1} - y_t\) satisfies (5.23)

It is convenient for us to introduce the following notation:

- \(\tau_t = \nu t\), a linear, deterministic trend
- \(m_t = \sum_{j=1}^t H z_j\), a martingale with time \(t + 1\) increment \(H z_{t+1}\)
- \(s_t = gx_t\), an (asymptotically) stationary component

We want to characterize and simulate components \(\tau_t, m_t, s_t\) of the decomposition.

A convenient way to do this is to construct an appropriate instance of a linear state space system by using LSS from QuantEcon.jl

This will allow us to use the routines in LSS to study dynamics

To start, observe that, under the dynamics in (5.22) and (5.23) and with the definitions just given,

\[
\begin{bmatrix}
1 \\
t + 1 \\
x_{t+1} \\
y_{t+1} \\
m_{t+1}
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 \\
0 & 0 & A & 0 & 0 \\
\nu & 0 & D' & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
x_t \\
t \\
x_t \\
y_t \\
m_t
\end{bmatrix} + \begin{bmatrix}
0 \\
0 \\
B \\
F' \\
H'
\end{bmatrix} z_{t+1}
\]

and

\[
\begin{bmatrix}
x_t \\
y_t \\
\tau_t \\
m_t \\
s_t
\end{bmatrix} = \begin{bmatrix}
0 & 0 & I & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & \nu & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & -g & 0 & 0
\end{bmatrix} \begin{bmatrix}
1 \\
t \\
x_t \\
y_t \\
m_t
\end{bmatrix}
\]

With

\[
\tilde{x} := \begin{bmatrix}
1 \\
x_t \\
y_t \\
m_t
\end{bmatrix} \quad \text{and} \quad \tilde{y} := \begin{bmatrix}
x_t \\
y_t \\
\tau_t \\
m_t \\
s_t
\end{bmatrix}
\]

we can write this as the linear state space system

\[
\tilde{x}_{t+1} = \tilde{A} \tilde{x}_t + \tilde{B} z_{t+1} \\
\tilde{y}_t = \tilde{D} \tilde{x}_t
\]

By picking out components of \(\tilde{y}_t\), we can track all variables of interest
5.3.4 Code

The type AMF_LSS_VAR mentioned above does all that we want to study our additive functional.

In fact AMF_LSS_VAR does more, as we shall explain below.

(A hint that it does more is the name of the type – here AMF stands for additive and multiplicative functional – the code will do things for multiplicative functionals too.)

Let’s use this code (embedded above) to explore the example process described above.

If you run the code that first simulated that example again and then the method call

```julia
plot_additive(amf, T)
```

You will generate (modulo randomness) the plot.

![Additive decomposition of \( y_t \)](image)

When we plot multiple realizations of a component in the 2nd, 3rd, and 4th panels, we also plot population 95% probability coverage sets computed using the LSS type.

We have chosen to simulate many paths, all starting from the same nonrandom initial conditions \( x_0, y_0 \) (you can tell this from the shape of the 95% probability coverage shaded areas).

Notice tell-tale signs of these probability coverage shaded areas:

- the purple one for the martingale component \( m_t \) grows with \( \sqrt{t} \)
- the green one for the stationary component \( s_t \) converges to a constant band
An associated multiplicative functional

Where \( \{y_t\} \) is our additive functional, let \( M_t = \exp(y_t) \)

As mentioned above, the process \( \{M_t\} \) is called a **multiplicative functional**

Corresponding to the additive decomposition described above we have the multiplicative decomposition of the \( M_t \)

\[
\frac{M_t}{M_0} = \exp(t\nu) \exp\left( \sum_{j=1}^{t} H \cdot Z_j \right) \exp\left( D'(I - A)^{-1}x_0 - D'(I - A)^{-1}x_t \right)
\]

or

\[
\frac{M_t}{M_0} = \exp(\tilde{\nu}t) \left( \frac{\tilde{M}_t}{M_0} \right) \left( \frac{\tilde{e}(X_0)}{\tilde{e}(x_t)} \right)
\]

where

\[
\tilde{\nu} = \nu + \frac{H \cdot H}{2}, \quad \tilde{M}_t = \exp\left( \sum_{j=1}^{t} \left( H \cdot z_j - \frac{H \cdot H}{2} \right) \right), \quad \tilde{M}_0 = 1
\]

and

\[
\tilde{e}(x) = \exp[g(x)] = \exp\left[ D'(I - A)^{-1}x \right]
\]

An instance of type AMF_LSS_VAR includes this associated multiplicative functional as an attribute

Let's plot this multiplicative functional for our example

If you run *the code that first simulated that example* again and then the method call

```plaintext
plot_multiplicative(amf, T)
```
As before, when we plotted multiple realizations of a component in the 2nd, 3rd, and 4th panels, we also plotted population 95% confidence bands computed using the LSS type.

Comparing this figure and the last also helps show how geometric growth differs from arithmetic growth.

**A peculiar large sample property**

Hansen and Sargent [*HS17*] (ch. 6) note that the martingale component $\tilde{M}_t$ of the multiplicative decomposition has a peculiar property:

- While $E_0\tilde{M}_t = 1$ for all $t \geq 0$, nevertheless . . .
- As $t \to +\infty$, $\tilde{M}_t$ converges to zero almost surely

The following simulation of many paths of $\tilde{M}_t$ illustrates this property:

```julia
grand(10021987)
pplot_martingales(amf, 12000)
```

Here's the resulting figure:

---

5.3. Additive Functionals

823
5.4 Multiplicative Functionals

Contents

- Multiplicative Functionals
  - Overview
  - A Log-Likelihood Process
  - Benefits from Reduced Aggregate Fluctuations

Co-authors: Chase Coleman and Balint Szoke

5.4.1 Overview

This lecture is a sequel to the lecture on additive functionals.

That lecture

1. defined a special class of additive functionals driven by a first-order vector VAR
2. by taking the exponential of that additive functional, created an associated multiplicative functional

This lecture uses this special class to create and analyze two examples

- A log likelihood process, an object at the foundation of both frequentist and Bayesian approaches to statistical inference
5.4.2 A Log-Likelihood Process

Consider a vector of additive functionals \( \{y_t\}_{t=0}^{\infty} \) described by

\[
\begin{align*}
x_{t+1} &= Ax_t + Bz_{t+1} \\
y_{t+1} - y_t &= Dx_t + Fz_{t+1},
\end{align*}
\]

where \( A \) is a stable matrix, \( \{z_{t+1}\}_{t=0}^{\infty} \) is an i.i.d. sequence of \( N(0, I) \) random vectors, \( F \) is nonsingular, and \( x_0 \) and \( y_0 \) are vectors of known numbers.

Evidently,

\[
x_{t+1} = (A - BF^{-1}D)x_t + BF^{-1}(y_{t+1} - y_t),
\]

so that \( x_{t+1} \) can be constructed from observations on \( \{y_s\}_{s=0}^{t+1} \) and \( x_0 \).

The distribution of \( y_{t+1} - y_t \) conditional on \( x_t \) is normal with mean \( Dx_t \) and nonsingular covariance matrix \( FF' \).

Let \( \theta \) denote the vector of free parameters of the model.

These parameters pin down the elements of \( A, B, D, F \).

The log likelihood function of \( \{y_s\}_{s=1}^{t} \) is

\[
\log L_t(\theta) = -\frac{1}{2} \sum_{j=1}^{t} (y_j - y_{j-1} - Dx_{j-1})' (FF')^{-1} (y_j - y_{j-1} - Dx_{j-1}) - \frac{kt}{2} \log(2\pi)
\]

Let's consider the case of a scalar process in which \( A, B, D, F \) are scalars and \( z_{t+1} \) is a scalar stochastic process.

We let \( \theta_0 \) denote the true values of \( \theta \), meaning the values that generate the data.

For the purposes of this exercise, set \( \theta_0 = (A, B, D, F) = (0.8, 1, 0.5, 0.2) \).

Set \( x_0 = y_0 = 0 \).

Simulating sample paths

Let's write a program to simulate sample paths of \( \{x_t, y_t\}_{t=0}^{\infty} \).

We'll do this by formulating the additive functional as a linear state space model and putting the LSS type to work.
using QuantEcon

###
This type and method are written to transform a scalar additive functional into a linear state space system.
###

```julia
function AMF_LSS_VAR(A::Real, B::Real, D::Real, F::Real = 0.0, \nu::Real = 0.0)
    # Construct BIG state space representation
    lss = construct_ss(A, B, D, F, \nu)
    return AMF_LSS_VAR(A, B, D, F, \nu, lss)
end

function construct_ss(A::Real, B::Real, D::Real, F::Real, \nu::Real)
    H, \nu = additive_decomp(A, B, D, F, \nu)
    # Build A matrix for LSS
    # Order of states is: [1, t, xt, yt, mt]
    A1 = [1 0 0 0 0] \quad \text{# Transition for } l
    A2 = [1 1 0 0 0] \quad \text{# Transition for } t
    A3 = [0 0 A 0 0] \quad \text{# Transition for } x_{t+1}
    A4 = [\nu 0 D 1 0] \quad \text{# Transition for } y_{t+1}
    A5 = [0 0 0 0 1] \quad \text{# Transition for } m_{t+1}
    Abar = vcat(A1, A2, A3, A4, A5)
    # Build B matrix for LSS
    Bbar = [0, 0, B, F, H]
    # Build G matrix for LSS
    # Order of observation is: [xt, yt, mt, st, tt]
    G1 = [0 0 1 0 0] \quad \text{# Selector for } x_{t}
    G2 = [0 0 0 1 0] \quad \text{# Selector for } y_{t}
    G3 = [0 0 0 0 1] \quad \text{# Selector for martingale}
    G4 = [0 0 -g 0 0] \quad \text{# Selector for stationary}
```

---

826 Chapter 5. Time Series Models
G5 = [0 ν 0 0 0]  # Selector for trend
Gbar = vcat(G1, G2, G3, G4, G5)

# Build LSS type
x0 = [0, 0, 0, 0, 0]
S0 = zeros(5, 5)
1ss = LSS(Abar, Bbar, Gbar, mu_0=x0, Sigma_0=S0)

return 1ss
end

***
Return values for the martingale decomposition (Proposition 4.3.3.)
- `H` : coefficient for the (linear) martingale,
- `kappa_a` : coefficient for the stationary component g(x)
***

function additive_decomp(A::Real, B::Real, D::Real, F::Real)
  A_res = 1 / (1 - A)
  g = D * A_res
  H = F + D * A_res * B

  return H, g
end

***
Return values for the multiplicative decomposition (Example 5.4.4.)
- `nu_tilde` : eigenvalue
- `H` : vector for the Jensen term
***

function multiplicative_decomp(A::Real, B::Real, D::Real, F::Real, ν::Real)
  H, g = additive_decomp(A, B, D, F)
  ν_tilde = ν + 0.5 * H^2

  return ν_tilde, H, g
end

function loglikelihood_path(amf::AMF_LSS_VAR, x::Vector, y::Vector)
  T = length(y)
  FF = F^2
  FFinv = 1/FF
  obs = temp .* FFinv .* temp
  obssum = cumsum(obs)
  scalar = (log(FF) + log(2pi)) .* collect(1:T-1)

  return (-0.5) * (obssum + scalar)
end

function loglikelihood(amf::AMF_LSS_VAR, x::Vector, y::Vector)
llh = loglikelihood_path(amf, x, y)
return llh[end]

The heavy lifting is done inside the AMF_LSS_VAR type

The following code adds some simple functions that make it straightforward to generate sample paths from an instance of AMF_LSS_VAR

```julia
function simulate_xy(amf::AMF_LSS_VAR, T::Integer)
    foo, bar = simulate(amf.lss, T)
    x = bar[1, :]
    y = bar[2, :]
    return x, y
end

function simulate_paths(amf::AMF_LSS_VAR, T::Integer=150, I::Integer=5000)
    # Allocate space
    storeX = Array{AbstractFloat}(I, T)
    storeY = Array{AbstractFloat}(I, T)

    for i in 1:I
        # Do specific simulation
        x, y = simulate_xy(amf, T)

        # Fill in our storage matrices
        storeX[i, :) = x
        storeY[i, :) = y
    end

    return storeX, storeY
end

function population_means(amf::AMF_LSS_VAR, T::Integer=150)
    # Allocate space
    xmean = Vector{AbstractFloat}(T)
    ymean = Vector{AbstractFloat}(T)

    # Pull out moment generator
    moment_generator = moment_sequence(amf.lss)
    state = start(moment_generator)
    for tt = 1:T
        tmoms, state = next(moment_generator, state)
        ymeans = tmoms[2]
    end
```
Now that we have these functions in our tool kit, let's apply them to run some simulations.

In particular, let's use our program to generate $I = 5000$ sample paths of length $T = 150$, labeled $\{x_i, y_i\}_{i=0}^\infty$ for $i = 1, \ldots, I$.

Then we compute averages of $\frac{1}{I} \sum_i x_i^i$ and $\frac{1}{I} \sum_i y_i^i$ across the sample paths and compare them with the population means of $x_t$ and $y_t$.

Here goes:

```julia
using PyPlot

A, B, D, F = 0.8, 1.0, 0.5, 0.2
amf = AMF_LSS_VAR(A, B, D, F)

T = 150
I = 5000

# Simulate and compute sample means
Xit, Yit = simulate_paths(amf, T, I)
Xmean_t = mean(Xit, 1)
Ymean_t = mean(Yit, 1)

# Compute population means
Xmean_pop, Ymean_pop = population_means(amf, T)

# Plot sample means vs population means
fig, ax = subplots(2, figsize=(14, 8))

ax[1][:plot](Xmean_t',
    label=L"\frac{1}{I}\sum_i x_t^i", color="b")
ax[1][:plot](Xmean_pop,
    label=L"\mathbb{E} x_t^\prime", color="k")
ax[1][:set_title](L"x_t")
ax[1][:set_xlim](0, T)
ax[1][:legend](loc=0)

ax[2][:plot](Ymean_t',
    label=L"\frac{1}{I}\sum_i y_t^i", color="b")
ax[2][:plot](Ymean_pop,
    label=L"\mathbb{E} y_t^\prime", color="k")
ax[2][:set_title](L"y_t")
ax[2][:set_xlim](0, T)
ax[2][:legend](loc=0)
```

Here's the resulting figure.
Simulating log-likelihoods

Our next aim is to write a program to simulate \( \{ \log L_t \mid \theta_o \}_{t=1}^T \). We want as inputs to this program the same sample paths \( \{x_t^i, y_t^i\}_{t=0}^T \) that we have already computed.

We now want to simulate \( I = 5000 \) paths of \( \{ \log L_t^i \mid \theta_o \}_{t=1}^T \):

- For each path, we compute \( \log L_T^i / T \)
- We also compute \( \frac{1}{T} \sum_{i=1}^{I} \log L_T^i / T \)

Then we to compare these objects.

Below we plot the histogram of \( \log L_T^i / T \) for realizations \( i = 1, \ldots, 5000 \)

```julia
function simulate_likelihood(amf::AMF_LSS_VAR, Xit::Array, Yit::Array)
    # Get size
    I, T = size(Xit)

    # Allocate space
    LLit = Array{Real}(I, T-1)

    for i in 1:I
        LLit[i, :] = loglikelihood_path(amf, Xit[i, :], Yit[i, :])
    end

    return LLit
end
```
# Get likelihood from each path $x^{(i)}, Y^{(i)}$

```python
LLit = simulate_likelihood(amf, Xit, Yit)

LLT = 1/T * LLit[:, end]
LLmean_t = mean(LLT)

fig, ax = subplots()
ax[hist](LLT)
ax[vlines](LLmean_t, ymin=0, ymax=I//3,
          color="k", linestyle="--", alpha=0.6)
fig[suptitle]($\text{Distribution of } \frac{1}{T} \log L_T \mid \theta_0$, $\rightarrow$ fontsize=14)
```

Here’s the resulting figure

Notice that the log likelihood is almost always nonnegative, implying that $L_T$ is typically bigger than 1

Recall that the likelihood function is a pdf (probability density function) and not a probability measure, so it can take values larger than 1

In the current case, the conditional variance of $\Delta y_{t+1}$, which equals $FF^T = 0.04$, is so small that the maximum value of the pdf is 2 (see the figure below)

This implies that approximately 75% of the time (a bit more than one sigma deviation), we should expect the increment of the log likelihood to be nonnegative

Let’s see this in a simulation

5.4. Multiplicative Functionals
using Distributions

normdist = Normal(0, F)
mult = 1.175
println("The pdf at +/- $mult sigma takes the value: $(pdf(normdist, mult*F))")
println("Probability of dL being larger than 1 is approx: $(cdf(normdist, mult*F) - cdf(normdist, -mult*F))")

# Compare this to the sample analogue:
L_increment = LLit[:,2:end] - LLit[:,1:end-1]
r,c = size(L_increment)
frac_nonnegative = sum(L_increment.>=0)/(c*r)
println("Fraction of dlogL being nonnegative in the sample is: $(frac_nonnegative)")

Heres the output

The pdf at +/- 1.175 sigma takes the value: 1.0001868966924388
Probability of dL being larger than 1 is approx: 0.7600052842019751
Fraction of dlogL being nonnegative in the sample is: 0.7601783783783784

Lets also plot the conditional pdf of $\Delta y_{t+1}$

xgrid = linspace(-1, 1, 100)
plot(xgrid, pdf.(normdist, xgrid))
title(L"Conditional pdf $f(\Delta y_{t+1} \mid x_t)$")
println("The pdf at +/- one sigma takes the value: $(pdf(normdist, F))")

Heres the resulting figure
The pdf at +/- one sigma takes the value: 1.2098536225957168

An alternative parameter vector

Now consider alternative parameter vector \( \theta_1 = [A, B, D, F] = [0.9, 1.0, 0.55, 0.25] \)

We want to compute \( \{ \log L_t \mid \theta_1 \}_T = 1 \)

The \( x_t, y_t \) inputs to this program should be exactly the same sample paths \( \{ x_t^i, y_t^i \}_T = 0 \) that we computed above.

This is because we want to generate data under the \( \theta_0 \) probability model but evaluate the likelihood under the \( \theta_1 \) model.

So our task is to use our program to simulate \( I = 5000 \) paths of \( \{ \log L_t^i \mid \theta_1 \}_T = 1 \)

- For each path, compute \( \frac{T}{T} \log L_T^i \)
- Then compute \( \frac{1}{T} \sum_{i=1}^{I} \frac{T}{T} \log L_T^i \)

We want to compare these objects with each other and with the analogous objects that we computed above.

Then we want to interpret outcomes.

A function that we constructed can handle these tasks.

The only innovation is that we must create an alternative model to feed in.

We will creatively call the new model amf2.

We make three graphs:

- the first sets the stage by repeating an earlier graph
- the second contains two histograms of values of log likelihoods of the two models over the period \( T \)
- the third compares likelihoods under the true and alternative models.

Here’s the code:

```julia
# Create the second (wrong) alternative model
A2, B2, D2, F2 = [0.9, 1.0, 0.55, 0.25]  # parameters for \( \theta_1 \) closer to \( \theta_0 \)
amf2 = AMF_LSS_VAR(A2, B2, D2, F2)

# Get likelihood from each path \( x^*(i), y^*(i) \)
LLit2 = simulate_likelihood(amf2, Xit, Yit)
LLT2 = 1/(T-1) * LLit2[:, end]
LLmean_t2 = mean(LLT2)

fig, ax = subplots()
ax[:hist](LLT2)
ax[:vlines](LLmean_t2, ymin=0, ymax=1400,
           color="k", linestyle="--", alpha=0.6)
```

5.4. Multiplicative Functionals
The resulting figure looks like this

![Histogram of log-likelihoods](image)

The resulting figure looks like this

834 Chapter 5. Time Series Models
Now well plot the histogram of the difference in log likelihood ratio

```plaintext
LLT_diff = LLT - LLT2

fig, ax = subplots(figsize=(8, 6))

ax.hist(LLT_diff, bins=50)

fig.suptitle(r'$\frac{1}{T} \left[ \log (L_T | \theta_0) - \log (L_T | \theta_1) \right]$', fontsize=15)
```

The resulting figure is as follows

5.4. Multiplicative Functionals
Interpretation

These histograms of log likelihood ratios illustrate important features of likelihood ratio tests as tools for discriminating between statistical models

- The log likelihood is higher on average under the true model – obviously a very useful property
- Nevertheless, for a positive fraction of realizations, the log likelihood is higher for the incorrect than for the true model
- in these instances, a likelihood ratio test mistakenly selects the wrong model
- These mechanics underlie the statistical theory of mistake probabilities associated with model selection tests based on likelihood ratio

(In a subsequent lecture, well use some of the code prepared in this lecture to illustrate mistake probabilities)
5.4.3 Benefits from Reduced Aggregate Fluctuations

Now let's turn to a new example of multiplicative functionals. This example illustrates ideas in the literatures on

- **long-run risk** in the consumption-based asset pricing literature (e.g., \[BY04\], \[HHL08\], \[Han07\])
- **benefits of eliminating aggregate fluctuations** in representative agent macro models (e.g., \[Tal00\], \[Luc03\])

Let \( c_t \) be consumption at date \( t \geq 0 \). Suppose that \( \{\log c_t\}_{t=0}^{\infty} \) is an additive functional described by

\[
\log c_{t+1} - \log c_t = \nu + D \cdot x_t + F \cdot z_{t+1}
\]

where

\[
x_{t+1} = Ax_t + Bz_{t+1}
\]

Here \( \{z_{t+1}\}_{t=0}^{\infty} \) is an i.i.d. sequence of \( \mathcal{N}(0, I) \) random vectors.

A representative household ranks consumption processes \( \{c_t\}_{t=0}^{\infty} \) with a utility functional \( \{V_t\}_{t=0}^{\infty} \) that satisfies

\[
\log V_t - \log c_t = U \cdot x_t + u
\]

where

\[
U = \exp(-\delta) \left[ I - \exp(-\delta)A \right]^{-1} D
\]

and

\[
u = \frac{\exp(-\delta)}{1 - \exp(-\delta)} \nu + \frac{(1 - \gamma)}{2} \frac{\exp(-\delta)}{1 - \exp(-\delta)} \left| D' \left[ I - \exp(-\delta)A \right]^{-1} B + F \right|^2.
\]

Here \( \gamma \geq 1 \) is a risk-aversion coefficient and \( \delta > 0 \) is a rate of time preference.

**Consumption as a multiplicative process**

We begin by showing that consumption is a **multiplicative functional** with representation

\[
\frac{c_t}{c_0} = \exp(\tilde{\nu}t) \left( \frac{\tilde{M}_t}{\tilde{M}_0} \right) \left( \frac{\tilde{e}(x_0)}{\tilde{e}(x_t)} \right)
\]

where \( \left( \frac{\tilde{M}_t}{\tilde{M}_0} \right) \) is a likelihood ratio process and \( \tilde{M}_0 = 1 \).
At this point, as an exercise, we ask the reader please to verify the following formulas for $\tilde{\nu}$ and $\tilde{e}(x_t)$ as functions of $A, B, D, F$:

$$\tilde{\nu} = \nu + \frac{H \cdot H}{2}$$

and

$$\tilde{e}(x) = \exp[g(x)] = \exp[D'(I - A)^{-1}x]$$

**Simulating a likelihood ratio process again**

Next, we want a program to simulate the likelihood ratio process $\{\tilde{M}_t\}_{t=0}^{\infty}$

In particular, we want to simulate 5000 sample paths of length $T = 1000$ for the case in which $x$ is a scalar and $[A, B, D, F] = [0.8, 0.001, 1.0, 0.01]$ and $\nu = 0.005$

After accomplishing this, we want to display a histogram of $\tilde{M}_T$ for $T = 1000$

Here is code that accomplishes these tasks

```julia
function simulate_martingale_components(amf::AMF_LSS_VAR,
                                        T::Integer=1000,
                                        I::Integer=5000)
    # Get the multiplicative decomposition
    $\nu, H, g = \text{multiplicative_decomp}(amf.A, amf.B, amf.D, amf.F, amf.}\nu$

    # Allocate space
    add_mart_comp = Array{Real}(I, T)

    # Simulate and pull out additive martingale component
    for i in 1:I
        foo, bar = simulate(amf.lss, T)
        # Martingale component is third component
        add_mart_comp[i, :] = bar[3, :]
    end

    mul_mart_comp =
        exp.(add_mart_comp' .- (collect(0:T-1) * H^2) / 2)'

    return add_mart_comp, mul_mart_comp
end

# Build model
amf_2 = AMF_LSS_VAR(0.8, 0.001, 1.0, 0.01, .005)

amc, mmc =
    simulate_martingale_components(amf_2, 1000, 5000)

amcT = amc[:, end]
mmcT = mmc[:, end]

println("The (min, mean, max) of additive Martingale component in period T is ", amcT, ", ", mmcT)
```

838 Chapter 5. Time Series Models
Here's the output:

The (min, mean, max) of additive Martingale component in period T is
-1.7419029969162607, -0.009316975586058086, 2.091259035641934

The (min, mean, max) of multiplicative Martingale component in period T is
0.15656398590834272, 0.9919363162991409, 7.234574417683094

Comments

- The preceding min, mean, and max of the cross-section of the date T realizations of the multiplicative martingale component of $c_t$ indicate that the sample mean is close to its population mean of 1
  - This outcome prevails for all values of the horizon T

- The cross-section distribution of the multiplicative martingale component of $c$ at date T approximates a log normal distribution well

- The histogram of the additive martingale component of $\log c_t$ at date T approximates a normal distribution well

Here's a histogram of the additive martingale component

```julia
fig, ax = subplots(figsize=(8, 6))
ax[:hist](amcT, bins=25, normed=true)
fig[:suptitle]("Histogram of Additive Martingale Component", fontsize=14)
```
Here's a histogram of the multiplicative martingale component

```python
fig, ax = subplots(figsize=(8, 6))
ax[:hist](mmcT, bins=25, normed=True)
fig[:suptitle]("Histogram of Multiplicative Martingale Component", fontsize=14)
```
Representing the likelihood ratio process

The likelihood ratio process $\{\widetilde{M}_t\}^\infty_{t=0}$ can be represented as

$$\widetilde{M}_t = \exp\left(\sum_{j=1}^{t} \left( H \cdot z_j - \frac{H \cdot H}{2}\right) \right), \quad \widetilde{M}_0 = 1,$$

where $H = [F + B'(I - A')^{-1}D]$

It follows that $\log \widetilde{M}_t \sim \mathcal{N}\left(-\frac{tH \cdot H}{2}, tH \cdot H\right)$ and that consequently $\widetilde{M}_t$ is log normal

Let's plot the probability density functions for $\log \widetilde{M}_t$ for $t = 100, 500, 1000, 10000, 100000$

Then let's use the plots to investigate how these densities evolve through time

We will plot the densities of $\log \widetilde{M}_t$ for different values of $t$

Note: scipy.stats.lognorm expects you to pass the standard deviation first $(tH \cdot H)$ and then the exponent of the mean as a keyword argument scale (scale=$\exp(-tH \cdot H/2)$)
• See the documentation here

This is peculiar, so make sure you are careful in working with the log normal distribution

Here is some code that tackles these tasks

```julia
function Mtilde_t_density(amf::AMF_LSS_VAR, t::Real;
    xmin::Real=1e-8,
    xmax::Real=5.0,
    npts::Integer=5000)

    # Pull out the multiplicative decomposition
    rtilde, H, g =
    H2 = H*H

    # The distribution
    mdist = LogNormal(-t * H2 / 2, sqrt(t * H2))
    x = linspace(xmin, xmax, npts)
    p = pdf.(mdist, x)

    return x, p
end

function logMtilde_t_density(amf::AMF_LSS_VAR, t::Real;
    xmin::Real=-15.0,
    xmax::Real=15.0,
    npts::Integer=5000)

    # Pull out the multiplicative decomposition
    rtilde, H, g =
    H2 = H*H

    # The distribution
    lmdist = Normal(-t * H2 / 2, sqrt(t * H2))
    x = linspace(xmin, xmax, npts)
    p = pdf.(lmdist, x)

    return x, p
end

times_to_plot =
    [10, 100, 500, 1000, 2500, 5000]
dens_to_plot =
    [Mtilde_t_density(amf_2, t, xmin=1e-8, xmax=6.0) for t in times_to_plot]
ddens_to_plot =
    [logMtilde_t_density(amf_2, t, xmin=-10.0, xmax=10.0) for t in times_to_plot]

fig, ax = subplots(3, 2, figsize=(8, 14))
ax = vec(ax)
```
fig[:sup_title](L"Densities of $\tilde{M}_t$", fontsize=18, y=1.02)
for (it, dens_t) in enumerate(dens_to_plot)
    x, pdf = dens_t
    ax[it][:set_title]("Density for time $\{\text{times_to_plot[it]}\}$")
    ax[it][:fill_between](x, zeros(pdf), pdf)
end

fig[:tight_layout]()
These probability density functions illustrate a peculiar property of log likelihood ratio processes:

- With respect to the true model probabilities, they have mathematical expectations equal to 1 for all $t \geq 0$
- They almost surely converge to zero

Welfare benefits of reduced random aggregate fluctuations

Suppose in the tradition of a strand of macroeconomics (for example Tallarini [Tal00], [Luc03]) we want to estimate the welfare benefits from removing random fluctuations around trend growth.

We shall compute how much initial consumption $c_0$ a representative consumer who ranks consumption streams according to (5.25) would be willing to sacrifice to enjoy the consumption stream

$$\frac{c_t}{c_0} = \exp(\tilde{\nu}t)$$

rather than the stream described by equation (5.26)

We want to compute the implied percentage reduction in $c_0$ that the representative consumer would accept.

To accomplish this, we write a function that computes the coefficients $U$ and $u$ for the original values of $A, B, D, F, \nu$, but also for the case that $A, B, D, F = [0, 0, 0, 0]$ and $\nu = \tilde{\nu}$.

Here's our code:

```julia
function Uu(amf::AMF_LSS_VAR, ::Real, ::Real)
    \nu_tilde, H, g = multiplicative_decomp(A, B, D, F, \nu

    resolv = 1 / (1 - \exp(-\delta) * A)
    vect = F + D * resolv * B

    U_risky = \exp(-\delta) * resolv * D
    u_risky = (\exp(-\delta) / (1 - \exp(-\delta)) ) * (\nu + 0.5 * (1 - \gamma) * (vect^2))

    U_det = 0
    u_det = (\exp(-\delta)/(1-exp(-\delta)) ) * \nu_tilde

    return U_risky, u_risky, U_det, u_det
end
```

# Set remaining parameters
\delta = 0.02
\gamma = 2.0

# Get coeffs
U_r, u_r, U_d, u_d = Uu(amf_2, \delta, \gamma)

The values of the two processes are

$$\log V_0^r = \log c_0^r + U^r x_0 + u^r$$
$$\log V_0^d = \log c_0^d + U^d x_0 + u^d$$

5.4. Multiplicative Functionals
We look for the ratio $\frac{c_r - c_d}{c_0}$ that makes $\log V^r_0 - \log V^d_0 = 0$

$$\log V^r_0 - \log V^d_0 + \log c'^r_0 - \log c'^d_0 = (U^r - U^d)x_0 + u^r - u^d$$

Hence, the implied percentage reduction in $c_0$ that the representative consumer would accept is given by

$$\frac{c'^r_0 - c'^d_0}{c'^r_0} = \exp \left( (U^r - U^d)x_0 + u^r - u^d \right)$$

Let's compute this

```plaintext
x0 = 0.0  # initial conditions
logVC_r = U_r * x0 + u_r
logVC_d = U_d * x0 + u_d

perc_reduct = 100 * (1 - exp(logVC_r - logVC_d))
perc_reduct
```

1.0809878812017448

We find that the consumer would be willing to take a percentage reduction of initial consumption equal to around 1.081

### Contents

- **Classical Control with Linear Algebra**
  - Overview
  - A Control Problem
  - Finite Horizon Theory
  - The Infinite Horizon Limit
  - Undiscounted Problems
  - Implementation
  - Exercises

### 5.5 Classical Control with Linear Algebra

#### 5.5.1 Overview

In an earlier lecture *Linear Quadratic Dynamic Programming Problems* we have studied how to solve a special class of dynamic optimization and prediction problems by applying the method of dynamic programming. In this class of problems
• the objective function is \textbf{quadratic} in states and controls
• the one-step transition function is \textbf{linear}
• shocks are i.i.d. Gaussian or martingale differences

In this lecture and a companion lecture \textit{Classical Filtering with Linear Algebra}, we study the classical theory of linear-quadratic (LQ) optimal control problems.

The classical approach does not use the two closely related methods – dynamic programming and Kalman filtering – that we describe in other lectures, namely, \textit{Linear Quadratic Dynamic Programming Problems} and \textit{A First Look at the Kalman Filter}

Instead they use either
• \textit{z}-transform and lag operator methods, or
• matrix decompositions applied to linear systems of first-order conditions for optimum problems.

In this lecture and the sequel \textit{Classical Filtering with Linear Algebra}, we mostly rely on elementary linear algebra

The main tool from linear algebra well put to work here is \textbf{LU decomposition}

Well begin with discrete horizon problems

Then well view infinite horizon problems as appropriate limits of these finite horizon problems

Later, we will examine the close connection between LQ control and least squares prediction and filtering problems

These classes of problems are connected in the sense that to solve each, essentially the same mathematics is used

\textbf{References}

Useful references include \text{[Whi63]}, \text{[HS80]}, \text{[Orf88]}, \text{[AP91]}, and \text{[Mut60]}

\textbf{5.5.2 A Control Problem}

Let $L$ be the \textbf{lag operator}, so that, for sequence $\{x_t\}$ we have $Lx_t = x_{t-1}$

More generally, let $L^kx_t = x_{t-k}$ with $L^0x_t = x_t$ and

$$d(L) = d_0 + d_1 L + \ldots + d_m L^m$$

where $d_0, d_1, \ldots, d_m$ is a given scalar sequence

Consider the discrete time control problem

$$
\max_{\{y_t\}} \lim_{N \to \infty} \sum_{t=0}^{N} \beta^t \left\{ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} [d(L)y_t]^2 \right\}, \quad (5.27)
$$

where
• $h$ is a positive parameter and $\beta \in (0, 1)$ is a discount factor

• $\{a_t\}_{t \geq 0}$ is a sequence of exponential order less than $\beta^{-1/2}$, by which we mean $\lim_{t \to \infty} \beta^{\frac{t}{2}} a_t = 0$

Maximization in (5.27) is subject to initial conditions for $y_{-1}, y_{-2}, \ldots, y_{-m}$

Maximization is over infinite sequences $\{y_t\}_{t \geq 0}$

**Example**

The formulation of the LQ problem given above is broad enough to encompass many useful models

As a simple illustration, recall that in *LQ Dynamic Programming Problems* we consider a monopolist facing stochastic demand shocks and adjustment costs

Let’s consider a deterministic version of this problem, where the monopolist maximizes the discounted sum

$$\sum_{t=0}^{\infty} \beta^t \pi_t$$

and

$$\pi_t = p_t q_t - c q_t - \gamma (q_{t+1} - q_t)^2 \quad \text{with} \quad p_t = \alpha_0 - \alpha_1 q_t + d_t$$

In this expression, $q_t$ is output, $c$ is average cost of production, and $d_t$ is a demand shock

The term $\gamma (q_{t+1} - q_t)^2$ represents adjustment costs

You will be able to confirm that the objective function can be rewritten as (5.27) when

• $a_t := \alpha_0 + d_t - c$

• $h := 2\alpha_1$

• $d(L) := \sqrt{\gamma}(I - L)$

Further examples of this problem for factor demand, economic growth, and government policy problems are given in ch. IX of *Sar87*

**5.5.3 Finite Horizon Theory**

We first study a finite $N$ version of the problem

Later we will study an infinite horizon problem solution as a limiting version of a finite horizon problem

(This will require being careful because the limits as $N \to \infty$ of the necessary and sufficient conditions for maximizing finite $N$ versions of (5.27) are not sufficient for maximizing (5.27))

We begin by

1. fixing $N > m$,

2. differentiating the finite version of (5.27) with respect to $y_0, y_1, \ldots, y_N$, and

3. setting these derivatives to zero
For $t = 0, \ldots, N - m$ these first-order necessary conditions are the *Euler equations*

For $t = N - m + 1, \ldots, N$, the first-order conditions are a set of *terminal conditions*

Consider the term

$$J = \sum_{t=0}^{N} \beta^t [d(L)y_t] [d(L)y_t]$$

$$= \sum_{t=0}^{N} \beta^t \left( d_0 y_t + d_1 y_{t-1} + \cdots + d_m y_{t-m} \right) \left( d_0 y_t + d_1 y_{t-1} + \cdots + d_m y_{t-m} \right)$$

Differentiating $J$ with respect to $y_t$ for $t = 0, 1, \ldots, N - m$ gives

$$\frac{\partial J}{\partial y_t} = 2\beta^t d_0 d(L)y_t + 2\beta^{t+1} d_1 d(L)y_{t+1} + \cdots + 2\beta^{t+m} d_m d(L)y_{t+m}$$

$$= 2\beta^t \left( d_0 + d_1 \beta L^{-1} + d_2 \beta^2 L^{-2} + \cdots + d_m \beta^m L^{-m} \right) d(L)y_t$$

We can write this more succinctly as

$$\frac{\partial J}{\partial y_t} = 2\beta^t d(\beta L^{-1}) d(L)y_t \quad (5.28)$$

Differentiating $J$ with respect to $y_t$ for $t = N - m + 1, \ldots, N$ gives

$$\frac{\partial J}{\partial y_N} = 2\beta^N d_0 d(L)y_N$$

$$\frac{\partial J}{\partial y_{N-1}} = 2\beta^{N-1} \left[ d_0 + \beta d_1 L^{-1} \right] d(L)y_{N-1}$$

$$\vdots$$

$$\frac{\partial J}{\partial y_{N-m+1}} = 2\beta^{N-m+1} \left[ d_0 + \beta L^{-1} d_1 + \cdots + \beta^{m-1} L^{-m+1} d_{m-1} \right] d(L)y_{N-m+1} \quad (5.29)$$

With these preliminaries under our belts, we are ready to differentiate (5.27)

Differentiating (5.27) with respect to $y_t$ for $t = 0, \ldots, N - m$ gives the Euler equations

$$\left[ h + d(\beta L^{-1}) d(L) \right] y_t = a_t, \quad t = 0, 1, \ldots, N - m \quad (5.30)$$

The system of equations (5.30) form a $2 \times m$ order linear *difference equation* that must hold for the values of $t$ indicated.

Differentiating (5.27) with respect to $y_t$ for $t = N - m + 1, \ldots, N$ gives the terminal conditions
\[ \beta^N (a_N - h y_N - d_0 d(L) y_N) = 0 \]
\[ \beta^{N-1} \left( a_{N-1} - h y_{N-1} - \left( d_0 + \beta d_1 L^{-1} \right) d(L) y_{N-1} \right) = 0 \]
\[ \vdots \]
\[ \beta^{N-m+1} \left( a_{N-m+1} - h y_{N-m+1} - (d_0 + \beta L^{-1} d_1 + \cdots + \beta^{m-1} L^{-m+1} d_{m-1}) d(L) y_{N-m+1} \right) = 0 \]

(5.31)

In the finite \( N \) problem, we want simultaneously to solve (5.30) subject to the \( m \) initial conditions \( y_{-1}, \ldots, y_{-m} \) and the \( m \) terminal conditions (5.31)

These conditions uniquely pin down the solution of the finite \( N \) problem

That is, for the finite \( N \) problem, conditions (5.30) and (5.31) are necessary and sufficient for a maximum, by concavity of the objective function

Next we describe how to obtain the solution using matrix methods

**Matrix Methods**

Let's look at how linear algebra can be used to tackle and shed light on the finite horizon LQ control problem

**A Single Lag Term**

Let's begin with the special case in which \( m = 1 \)

We want to solve the system of \( N + 1 \) linear equations

\[
[h + d(\beta L^{-1}) d(L)] y_t = a_t, \quad t = 0, 1, \ldots, N - 1
\]
\[
\beta^N [a_N - h y_N - d_0 d(L) y_N] = 0
\]

(5.32)

where \( d(L) = d_0 + d_1 L \)

These equations are to be solved for \( y_0, y_1, \ldots, y_N \) as functions of \( a_0, a_1, \ldots, a_N \) and \( y_{-1} \)

Let

\[
\phi(L) = \phi_0 + \phi_1 L + \beta \phi_1 L^{-1} = h + d(\beta L^{-1}) d(L) = (h + d_0^2 + d_1^2) + d_1 d_0 L + d_1 d_0 \beta L^{-1}
\]

Then we can represent (5.32) as the matrix equation

\[
\begin{bmatrix}
\phi_0 & 0 & 0 & \ldots & 0 \\
\beta \phi_1 & \phi_0 & \phi_1 & \ldots & 0 \\
0 & \beta \phi_1 & \phi_0 & \ldots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & \ldots & \ldots & \beta \phi_1 & \phi_0 & \phi_1 \\
0 & \ldots & \ldots & \beta \phi_1 & \phi_0 \\
\end{bmatrix}
\begin{bmatrix}
y_N \\
y_{N-1} \\
y_{N-2} \\
\vdots \\
y_1 \\
y_0 \\
\end{bmatrix}
= \begin{bmatrix}
a_N \\
a_{N-1} \\
a_{N-2} \\
\vdots \\
a_1 \\
a_{0} - \phi_1 y_{-1} \\
\end{bmatrix}
\]

(5.33)
or

\[ W \ddot{y} = \ddot{a} \]  

(5.34)

Notice how we have chosen to arrange the \( y_t \)s in reverse time order.

The matrix \( W \) on the left side of (5.33) is almost a Toeplitz matrix (where each descending diagonal is constant).

There are two sources of deviation from the form of a Toeplitz matrix:

1. The first element differs from the remaining diagonal elements, reflecting the terminal condition.
2. The subdiagonal elements equal \( \beta \) time the superdiagonal elements.

The solution of (5.34) can be expressed in the form

\[ \ddot{y} = W^{-1} \ddot{a} \]  

(5.35)

which represents each element \( y_t \) of \( \ddot{y} \) as a function of the entire vector \( \ddot{a} \).

That is, \( y_t \) is a function of past, present, and future values of \( a \), as well as of the initial condition \( y_{-1} \).

**An Alternative Representation**

An alternative way to express the solution to (5.33) or (5.34) is in so called feedback-feedforward form.

The idea here is to find a solution expressing \( y_t \) as a function of past \( y \)s and current and future \( a \)s.

To achieve this solution, one can use an LU decomposition of \( W \).

There always exists a decomposition of \( W \) of the form \( W = LU \) where

- \( L \) is an \((N + 1) \times (N + 1)\) lower triangular matrix.
- \( U \) is an \((N + 1) \times (N + 1)\) upper triangular matrix.

The factorization can be normalized so that the diagonal elements of \( U \) are unity.

Using the LU representation in (5.35), we obtain

\[ U \ddot{y} = L^{-1} \ddot{a} \]  

(5.36)

Since \( L^{-1} \) is lower triangular, this representation expresses \( y_t \) as a function of

- lagged \( y \)s (via the term \( U \ddot{y} \)), and
- current and future \( a \)s (via the term \( L^{-1} \ddot{a} \)).

Because there are zeros everywhere in the matrix on the left of (5.33) except on the diagonal, superdiagonal, and subdiagonal, the \( LU \) decomposition takes
• $L$ to be zero except in the diagonal and the leading subdiagonal

• $U$ to be zero except on the diagonal and the superdiagonal

Thus, (5.36) has the form

$$
\begin{bmatrix}
1 & U_{12} & 0 & \cdots & 0 & 0 \\
0 & 1 & U_{23} & \cdots & 0 & 0 \\
0 & 0 & 1 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & 1 & U_{N,N+1} \\
0 & 0 & 0 & \cdots & 0 & 1
\end{bmatrix}
\begin{bmatrix}
y_N \\
y_{N-1} \\
y_{N-2} \\
y_{N-3} \\
\vdots \\
y_1 \\
y_0
\end{bmatrix}
= 
\begin{bmatrix}
L_{11}^{-1} & 0 & 0 & \cdots & 0 \\
L_{21}^{-1} & L_{22}^{-1} & 0 & \cdots & 0 \\
L_{31}^{-1} & L_{32}^{-1} & L_{33}^{-1} & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
L_{N,1}^{-1} & L_{N,2}^{-1} & L_{N,3}^{-1} & \cdots & L_{N,N+1}^{-1}
\end{bmatrix}
\begin{bmatrix}
a_N \\
a_{N-1} \\
a_{N-2} \\
a_1 \\
\vdots \\
\phi_1 y_{-1} - a_0
\end{bmatrix}
$$

where $L_{ij}^{-1}$ is the $(i, j)$ element of $L^{-1}$ and $U_{ij}$ is the $(i, j)$ element of $U$.

Note how the left side for a given $t$ involves $y_t$ and one lagged value $y_{t-1}$ while the right side involves all future values of the forcing process $a_t, a_{t+1}, \ldots, a_N$.

**Additional Lag Terms**

We briefly indicate how this approach extends to the problem with $m > 1$

Assume that $\beta = 1$ and let $D_{m+1}$ be the $(m+1) \times (m+1)$ symmetric matrix whose elements are determined from the following formula:

$$
D_{jk} = d_0 d_{k-j} + d_1 d_{k-j+1} + \cdots + d_{j-1} d_{k-1}, \quad k \geq j
$$

Let $I_{m+1}$ be the $(m + 1) \times (m + 1)$ identity matrix.

Let $\phi_j$ be the coefficients in the expansion $\phi(L) = h + d(L^{-1})d(L)$

Then the first order conditions (5.30) and (5.31) can be expressed as:

$$
(D_{m+1} + h I_{m+1})
\begin{bmatrix}
y_N \\
y_{N-1} \\
\vdots \\
y_{N-m}
\end{bmatrix}
= 
\begin{bmatrix}
a_N \\
a_{N-1} \\
\vdots \\
a_{N-m}
\end{bmatrix}
+ M
\begin{bmatrix}
y_{N-m+1} \\
y_{N-m} \\
\vdots \\
y_{N-2m}
\end{bmatrix}
$$

where $M$ is $(m + 1) \times m$ and

$$
M_{ij} = \begin{cases} 
D_{i-j,m+1} & \text{for } i > j \\
0 & \text{for } i \leq j 
\end{cases}
$$
\[ \phi_m y_{N-1} + \phi_{m-1} y_{N-2} + \ldots + \phi_0 y_{N-m-1} + \phi_1 y_{N-m-2} + \ldots + \phi_m y_{N-2m-1} = a_{N-m-1} \]
\[ \phi_m y_{N-2} + \phi_{m-1} y_{N-3} + \ldots + \phi_0 y_{N-m-2} + \phi_1 y_{N-m-3} + \ldots + \phi_m y_{N-2m-2} = a_{N-m-2} \]
\[ \vdots \]
\[ \phi_m y_m + \phi_{m-1} y_{m-1} + \phi_{m-2} + \ldots + \phi_0 y_0 + \phi_1 y_{-1} + \ldots + \phi_m y_{-m} = a_1 \]
\[ \phi_m y_{m+1} + \phi_{m-1} y_m + \ldots + \phi_0 y_1 + \phi_1 y_0 + \phi_m y_{m+1} = a_0 \]

As before, we can express this equation as \( W \tilde{y} = \tilde{a} \)

The matrix on the left of this equation is almost Toeplitz, the exception being the leading \( m \times m \) sub matrix in the upper left hand corner

We can represent the solution in feedback-feedforward form by obtaining a decomposition \( LU = W \), and obtain

\[ U \tilde{y} = L^{-1} \tilde{a} \]

\[ \sum_{j=0}^{t} U_{t-N+1, -t-N+1+j+1} y_{t-j} = \sum_{j=0}^{N-t} L_{t-N+1, -t-N+1-j} \tilde{a}_{t+j}, \quad t = 0, 1, \ldots, N \]

where \( L_{t,s}^{-1} \) is the element in the \((t, s)\) position of \( L \), and similarly for \( U \)

The left side of equation (5.37) is the feedback part of the optimal control law for \( y_t \), while the right-hand side is the feedforward part

We note that there is a different control law for each \( t \)

Thus, in the finite horizon case, the optimal control law is time dependent

It is natural to suspect that as \( N \to \infty \), (5.37) becomes equivalent to the solution of our infinite horizon problem, which below we shall show can be expressed as

\[ c(L) y_t = c(\beta L^{-1})^{-1} a_t, \]

so that as \( N \to \infty \) we expect that for each fixed \( t \), \( U_{t-t}^{-1} \to c_j \) and \( L_{t,t+j} \) approaches the coefficient on \( L^{-1} \) in the expansion of \( c(\beta L^{-1})^{-1} \)

This suspicion is true under general conditions that we shall study later

For now, we note that by creating the matrix \( W \) for large \( N \) and factoring it into the \( LU \) form, good approximations to \( c(L) \) and \( c(\beta L^{-1})^{-1} \) can be obtained

### 5.5.4 The Infinite Horizon Limit

For the infinite horizon problem, we propose to discover first-order necessary conditions by taking the limits of (5.30) and (5.31) as \( N \to \infty \)
This approach is valid, and the limits of (5.30) and (5.31) as \( N \) approaches infinity are first-order necessary conditions for a maximum.

However, for the infinite horizon problem with \( \beta < 1 \), the limits of (5.30) and (5.31) are, in general, not sufficient for a maximum. That is, the limits of (5.31) do not provide enough information uniquely to determine the solution of the Euler equation (5.30) that maximizes (5.27).

As we shall see below, a side condition on the path of \( y_t \) that together with (5.30) is sufficient for an optimum is

\[
\sum_{t=0}^{\infty} \beta^t h y_t^2 < \infty \tag{5.38}
\]

All paths that satisfy the Euler equations, except the one that we shall select below, violate this condition and, therefore, evidently lead to (much) lower values of (5.27) than does the optimal path selected by the solution procedure below.

Consider the characteristic equation associated with the Euler equation

\[
h + d(\beta z^{-1}) d(z) = 0 \tag{5.39}
\]

Notice that if \( \tilde{z} \) is a root of equation (5.39), then so is \( \beta \tilde{z}^{-1} \).

Thus, the roots of (5.39) come in \( \beta \)-reciprocal pairs.

Assume that the roots of (5.39) are distinct.

Let the roots be, in descending order according to their moduli, \( z_1, z_2, \ldots, z_{2m} \).

From the reciprocal pairs property and the assumption of distinct roots, it follows that \( |z_j| > \sqrt{\beta} \) for \( j \leq m \) and \( |z_j| < \sqrt{\beta} \) for \( j > m \).

It also follows that \( z_{2m-j} = \beta z_{j+1}^{-1}, j = 0, 1, \ldots, m-1 \).

Therefore, the characteristic polynomial on the left side of (5.39) can be expressed as

\[
h + d(\beta z^{-1}) d(z) = z^{-m} z_0(z - z_1) \cdots (z - z_m)(z - z_{m+1}) \cdots (z - z_{2m})
\]

\[
= z^{-m} z_0(z - z_1)(z - z_2) \cdots (z - z_m)(z - \beta z_{m}^{-1}) \cdots (z - \beta z_{2m}^{-1})(z - \beta z_{1}^{-1}) \tag{5.40}
\]

where \( z_0 \) is a constant.

In (5.40), we substitute \( (z - z_j) = -z_j(1 - \frac{1}{z_j}z) \) and \( (z - \beta z_j^{-1}) = z(1 - \frac{\beta}{z_j} z^{-1}) \) for \( j = 1, \ldots, m \) to get

\[
h + d(\beta z^{-1}) d(z) = (-1)^m (z_0 z_1 \cdots z_m)(1 - \frac{1}{z_1}z) \cdots (1 - \frac{1}{z_m}z)(1 - \frac{1}{z_1} \beta z^{-1}) \cdots (1 - \frac{1}{z_m} \beta z^{-1})
\]

Now define \( c(z) = \sum_{j=0}^{m} c_j z^j \) as
\[ c(z) = \left[ (-1)^m z_0 z_1 \cdots z_m \right]^{1/2} \left( 1 - \frac{z}{z_1} \right) \left( 1 - \frac{z}{z_2} \right) \cdots \left( 1 - \frac{z}{z_m} \right) \]  

(5.41)

Notice that (5.40) can be written

\[ h + d(\beta z^{-1})d(z) = c(\beta z^{-1})c(z) \]  

(5.42)

It is useful to write (5.41) as

\[ c(z) = c_0(1 - \lambda_1 z) \cdots (1 - \lambda_m z) \]  

(5.43)

where

\[ c_0 = \left[ (-1)^m z_0 z_1 \cdots z_m \right]^{1/2}; \quad \lambda_j = \frac{1}{z_j}, \quad j = 1, \ldots, m \]

Since \(|z_j| > \sqrt{\beta}\) for \(j = 1, \ldots, m\) it follows that \(|\lambda_j| < 1/\sqrt{\beta}\) for \(j = 1, \ldots, m\).

Using (5.43), we can express the factorization (5.42) as

\[ h + d(\beta z^{-1})d(z) = c_0^2(1 - \lambda_1 z) \cdots (1 - \lambda_m z)(1 - \lambda_1 \beta z^{-1}) \cdots (1 - \lambda_m \beta z^{-1}) \]

In sum, we have constructed a factorization (5.42) of the characteristic polynomial for the Euler equation in which the zeros of \(c(z)\) exceed \(\beta^{1/2}\) in modulus, and the zeros of \(c(\beta z^{-1})\) are less than \(\beta^{1/2}\) in modulus.

Using (5.42), we now write the Euler equation as

\[ c(\beta L^{-1})c(L)y_t = a_t \]

The unique solution of the Euler equation that satisfies condition (5.38) is

\[ c(L)y_t = c(\beta L^{-1})^{-1}a_t \]  

(5.44)

This can be established by using an argument paralleling that in chapter IX of [Sar87].

To exhibit the solution in a form paralleling that of [Sar87], we use (5.43) to write (5.44) as

\[ (1 - \lambda_1 L) \cdots (1 - \lambda_m L)y_t = \frac{c_0^{-2}a_t}{(1 - \beta \lambda_1 L^{-1}) \cdots (1 - \beta \lambda_m L^{-1})} \]  

(5.45)

Using partial fractions, we can write the characteristic polynomial on the right side of (5.45) as

\[ \sum_{j=1}^{m} \frac{A_j}{1 - \lambda_j \beta L^{-1}} \quad \text{where} \quad A_j := \frac{c_0^{-2}}{\prod_{i \neq j}(1 - \frac{\lambda_i}{\lambda_j})} \]
Then (5.45) can be written

\[
(1 - \lambda_1 L) \cdots (1 - \lambda_m L) y_t = \sum_{j=1}^{m} \frac{A_j}{1 - \lambda_j \beta L^{-1}} a_t
\]

or

\[
(1 - \lambda_1 L) \cdots (1 - \lambda_m L) y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} (\lambda_j \beta)^k a_{t+k}
\] (5.46)

Equation (5.46) expresses the optimum sequence for \( y_t \) in terms of \( m \) lagged \( y_s \), and \( m \) weighted infinite geometric sums of future \( a_t \)s.

Furthermore, (5.46) is the unique solution of the Euler equation that satisfies the initial conditions and condition (5.38).

In effect, condition (5.38) compels us to solve the unstable roots of \( h + d(\beta z^{-1})d(z) \) forward (see [Sar87]).

The step of factoring the polynomial \( h + d(\beta z^{-1})d(z) \) into \( c(\beta z^{-1})c(z) \), where the zeros of \( c(z) \) all have modulus exceeding \( \sqrt{\beta} \), is central to solving the problem.

We note two features of the solution (5.46):

- Since \( |\lambda_j| < 1/\sqrt{\beta} \) for all \( j \), it follows that \( (\lambda_j \beta) < \sqrt{\beta} \).
- The assumption that \( \{a_t\} \) is of exponential order less than \( 1/\sqrt{\beta} \) is sufficient to guarantee that the geometric sums of future \( a_t \)s on the right side of (5.46) converge.

We immediately see that those sums will converge under the weaker condition that \( \{a_t\} \) is of exponential order less than \( \phi^{-1} \) where \( \phi = \max \{\beta \lambda_i, i = 1, \ldots, m\} \).

Note that with \( a_t \) identically zero, (5.46) implies that in general \( |y_t| \) eventually grows exponentially at a rate given by \( \max_i |\lambda_i| \).

The condition \( \max_i |\lambda_i| < 1/\sqrt{\beta} \) guarantees that condition (5.38) is satisfied.

In fact, \( \max_i |\lambda_i| < 1/\sqrt{\beta} \) is a necessary condition for (5.38) to hold.

Were (5.38) not satisfied, the objective function would diverge to \( -\infty \), implying that the \( y_t \) path could not be optimal.

For example, with \( a_t = 0 \), for all \( t \), it is easy to describe a naive (nonoptimal) policy for \( \{y_t, t \geq 0\} \) that gives a finite value of (5.43).

We can simply let \( y_t = 0 \) for \( t \geq 0 \).

This policy involves at most \( m \) nonzero values of \( hy_t^2 \) and \( [d(L)y_t]^2 \), and so yields a finite value of (5.27).

Therefore it is easy to dominate a path that violates (5.38).

### 5.5.5 Undiscounted Problems

It is worthwhile focusing on a special case of the LQ problems above: the undiscounted problem that emerges when \( \beta = 1 \).
In this case, the Euler equation is
\[
\left( h + d(L^{-1})d(L) \right) y_t = a_t
\]
The factorization of the characteristic polynomial (5.42) becomes
\[
\left( h + d(z^{-1})d(z) \right) = c(z^{-1})c(z)
\]
where
\[
c(z) = c_0(1 - \lambda_1 z) \cdots (1 - \lambda_m z)
\]
\[
c_0 = \left[ (-1)^m z_0 z_1 \ldots z_m \right]
\]
\[
|\lambda_j| < 1 \text{ for } j = 1, \ldots, m
\]
\[
\lambda_j = \frac{1}{z_j} \text{ for } j = 1, \ldots, m
\]
\[
z_0 = \text{ constant}
\]
The solution of the problem becomes
\[
(1 - \lambda_1 L) \cdots (1 - \lambda_m L) y_t = \sum_{j=1}^m A_j \sum_{k=0}^{\infty} \lambda_j^k a_{t+k}
\]

**Transforming discounted to undiscounted problem**

Discounted problems can always be converted into undiscounted problems via a simple transformation.

Consider problem (5.27) with $0 < \beta < 1$

Define the transformed variables
\[
\tilde{a}_t = \beta^{t/2} a_t, \quad \tilde{y}_t = \beta^{t/2} y_t \tag{5.47}
\]

Then notice that $\beta^t [d(L)y_t]^2 = [\tilde{d}(L)\tilde{y}_t]^2$ with $\tilde{d}(L) = \sum_{j=0}^m \tilde{d}_j L^j$ and $\tilde{d}_j = \beta^{j/2} d_j$

Then the original criterion function (5.27) is equivalent to
\[
\lim_{N \to \infty} \sum_{t=0}^N \left\{ \tilde{a}_t \tilde{y}_t - \frac{1}{2} h \tilde{y}_t^2 - \frac{1}{2} [\tilde{d}(L)\tilde{y}_t]^2 \right\} \tag{5.48}
\]

which is to be maximized over sequences $\{\tilde{y}_t, t = 0, \ldots\}$ subject to $\tilde{y}_{-1}, \ldots, \tilde{y}_{-m}$ given and $\{\tilde{a}_t, t = 1, \ldots\}$ a known bounded sequence

The Euler equation for this problem is $[h + \tilde{d}(L^{-1})\tilde{d}(L)] \tilde{y}_t = \tilde{a}_t$

The solution is
\[
(1 - \tilde{\lambda}_1 L) \cdots (1 - \tilde{\lambda}_m L) \tilde{y}_t = \sum_{j=1}^m \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k}
\]
or

\[
\tilde{y}_t = \tilde{f}_1 \tilde{y}_{t-1} + \cdots + \tilde{f}_m \tilde{y}_{t-m} + \sum_{j=1}^m \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k},
\]

where \( \tilde{c}(z^{-1})\tilde{c}(z) = h + \tilde{d}(z^{-1})\tilde{d}(z) \), and where

\[
[(-1)^m \tilde{z}_0 \tilde{z}_1 \cdots \tilde{z}_m]^{1/2}(1 - \tilde{\lambda}_1 z) \cdots (1 - \tilde{\lambda}_m z) = \tilde{c}(z), \text{ where } |\tilde{\lambda}_j| < 1
\]

We leave it to the reader to show that (5.49) implies the equivalent form of the solution

\[
y_t = f_1 y_{t-1} + \cdots + f_m y_{t-m} + \sum_{j=1}^m A_j \sum_{k=0}^{\infty} \lambda_j^k \alpha_{t+k}
\]

where

\[
f_j = \tilde{f}_j \beta^{-j/2}, \quad A_j = \tilde{A}_j, \quad \lambda_j = \tilde{\lambda}_j \beta^{-1/2}
\]

The transformations (5.47) and the inverse formulas (5.50) allow us to solve a discounted problem by first solving a related undiscounted problem

### 5.5.6 Implementation

Code that computes solutions to the LQ problem using the methods described above can be found in file control_and_filter.jl

Here's how it looks

```julia
#=
Author: Shunsuke Hori
=

using Polynomials

struct LQFilter{Tf<:Real, Ti<:Integer, Tf<:AbstractFloat}
    d::Vector{Tf}
    h::Tf
    \_y_m::Vector{Tf}
    m::Ti
    \_\beta::Tf
    _r::Union{Vector{Tf},Void}
    k::Union{Ti,Void}
end
```

Chapter 5. Time Series Models
### Parameters
---------

d : Vector  
The order of the coefficients: \([d_0, d_1, \ldots, d_m]\)

h : Real  
Parameter of the objective function (corresponding to the quadratic term)

y_m : Vector  
Initial conditions for y

r : Vector  
The order of the coefficients: \([r_0, r_1, \ldots, r_k]\)  
(optional, if not defined -> deterministic problem)

\(\beta\) : Real or nothing  
Discount factor (optional, default value is one)

h_eps :  

---

```
function LQFilter{TR<:Real}(d::Vector{TR},
h::TR,
y_m::Vector{TR};
r::Union{Vector{TR},Void}=nothing,
\beta::Union{TR,Void}=nothing,
h_eps::Union{TR,Void}=nothing,
)

m = length(d) - 1

m == length(y_m) ||
    throw(ArgumentError("y_m and d must be of same length = $m"))

# Define the coefficients of up front
#-----------------------------------------------------

= Vector{TR}(2m + 1)
for i in -m:m
    [m-i+1] = sum(diag(d*d', -i))
end

[m+1] = [m+1] + h

# If r is given calculate the vector _r
#-----------------------------------------------------

if r == nothing
    k=nothing
    _r = nothing
else
    k = size(r, 1) - 1
    _r = Vector{TR}(2k + 1)
    for i = -k:k
```

5.5. Classical Control with Linear Algebra 859
```julia
    _r[k-i+1] = sum(diag(r*r', -i))

end

if h_eps != nothing
    _r[k+1] = _r[k+1] + h_eps
end

#-----------------------------------------------------
# If β is given, define the transformed variables
#-----------------------------------------------------
if β == nothing
    β = 1.0
else
    d = β.^((collect(0:m)/2) + d)
    y_m = y_m + β.^(-collect(1:m)/2)
end

return LQFilter(d, h, y_m, m, , β, _r, k)
end

""
This constructs the matrices W and W_m for a given number of periods N
""

function construct_W_and_Wm(lqf::LQFilter, N::Integer)

    d, m = lqf.d, lqf.m

    W = zeros(N + 1, N + 1)
    W_m = zeros(N + 1, m)

    #---------------------------------------
    # Terminal conditions
    #---------------------------------------

    D_m1 = zeros(m + 1, m + 1)
    M = zeros(m + 1, m)

    # (1) Construct the D_{m+1} matrix using the formula
    for j in 1:(m+1)
        for k in j:(m+1)
            D_m1[j, k] = dot(d[1:j, 1], d[k-j+1:k, 1])
        end
    end

    # Make the matrix symmetric
    D_m1 = D_m1 + D_m1' - diagm(diag(D_m1))

    # (2) Construct the M matrix using the entries of D_m1
    for j in 1:m
        for i in (j + 1):(m + 1)
```

860 Chapter 5. Time Series Models
M[i, j] = D_m1[i-j, m+1]

end
end M

# Euler equations for t = 0, 1, ..., N-(m+1)
#----------------------------------------------
# Calculate the roots of the 2m-polynomial
W[1:(m + 1), 1:(m + 1)] = D_m1 + h * eye(m + 1)
W[1:(m + 1), (m + 2):(2m + 1)] = M

for (i, row) in enumerate((m + 2):((N + 1 - m)))
    W[row, (i + 1):(2m + 1 + i)] = '
end

for i in 1:m
    W[N - m + i + 1, end-(2m + 1 - i)+1:end] = [1:end-1]
end

for i in 1:m
    W_m[N - i + 2, 1:(m - i)+1] = [(m + 1 + i):end]
end

return W, W_m

""
This function calculates z_0 and the 2m roots of the characteristic equation associated with the Euler equation (1.7)
Note:
------
\texttt{poly(roots)} from Polynomial.jll defines a polynomial using its roots that can be evaluated at any point by \texttt{polyval(Poly,x)}. If x_1, x_2, \ldots, x_m are the roots then \texttt{polyval(poly(roots),x)} = (x - x_1)(x - x_2)...(x - x_m)
""

function roots_of_characteristic(lqf::LQFilter)
    m, = lqf.m, lqf.

    # Calculate the roots of the 2m-polynomial
    _poly=Poly([end:-1:1])
    roots = roots(_poly)

    # sort the roots according to their length (in descending order)
    roots_sorted = sort(roots, by=abs)[end:-1:1]
    z_0 = sum() / polyval(poly(roots), 1.0)
    z_1_to_m = roots_sorted[1:m]  # we need only those outside the unit circle
    \lambda = 1 ./ z_1_to_m
    return z_1_to_m, z_0, \lambda
end

This function computes the coefficients \( \{c_j, j = 0, 1, \ldots, m\} \) for
\[
c(z) = \sum_{j = 0}^{m} c_j z^j
\]
Based on the expression (1.9). The order is
\[
c\_coeffs = [c_0, c_1, \ldots, c_{(m-1)}, c_m]
\]

def coefs_of_c(lqf::LQFilter)
    m = lqf.m
    z_1_to_m, z_0, \lambda = roots_of_characteristic(lqf)
    c_0 = (z_0 \times \text{prod}(z_1_to_m) \times (-1.0)^m)^{0.5}
    c\_coeffs = coefs(poly(z_1_to_m)) \times z_0 / c_0
    return c\_coeffs
end

This function calculates \( \{\lambda_j, j=1,\ldots,m\} \) and \( \{A_j, j=1,\ldots,m\} \)
of the expression (1.15)

def solution(lqf::LQFilter)
    z_1_to_m, z_0, \lambda = roots_of_characteristic(lqf)
    c_0 = coefs_of_c(lqf)
    A = zeros(lqf.m)
    for j in 1:m
        denom = 1 - \lambda / \lambda[j]
        A[j] = c_0^{(-2)} / \text{prod}(\text{denom}[1:m .!= j])
    end
    return \lambda, A
end

This function constructs the covariance matrix for \( x^N \) (see section 6)
for a given period \( N \)

def construct_V(lqf::LQFilter; N::Integer=nothing)
    if N == nothing
        error("N must be provided!!")
    end
    if !(typeof(N) <: Integer)
        throw(ArgumentError("N must be Integer!"))
    end

    _r, k = lqf._r, lqf.k
    V = zeros(N, N)
    for i in 1:N
        for j in 1:N
            if abs(i-j) <= k
                V[i, j] = _r[k + abs(i-j)+1]
            end
        end
    end
    return V
Assuming that the \( u \)'s are normal, this method draws a random path for \( x^N \)

```
function simulate_a(lqf::LQFilter, N::Integer)
    V = construct_V(N + 1)
    d = MVNSampler(zeros(N + 1), V)
    return rand(d)
end
```

This function implements the prediction formula discussed in section 6 (1.59)
It takes a realization for \( a^N \), and the period in which the prediction is formed.

Output: \( E[\bar{a}_t | a_t, a_{t-1}, ..., a_1, a_0] \)

```
function predict(lqf::LQFilter, a_hist::Vector, t::Integer)
    N = length(a_hist) - 1
    V = construct_V(N + 1)

    aux_matrix = zeros(N + 1, N + 1)
    aux_matrix[1:t+1 , 1:t+1 ] = eye(t + 1)
    L = chol(V)
    Ea_hist = inv(L) * aux_matrix * L * a_hist

    return Ea_hist
end
```

- if \( t \) is NOT given it takes \( a_{hist} \) (Vector or Array) as a deterministic \( a_t \)
- if \( t \) is given, it solves the combined control prediction problem (section 7)
  (by default, \( t = \) nothing -> deterministic)

for a given sequence of \( a_t \) (either deterministic or a particular realization), it calculates the optimal \( y_t \) sequence using the method of the lecture.

Note:
------
lufact normalizes \( L, U \) so that \( L \) has unit diagonal elements
To make things consistent with the lecture, we need an auxiliary diagonal matrix \( D \) which renormalizes \( L \) and \( U \)

```
function optimal_y(lqf::LQFilter, a_hist::Vector, t = nothing)
    \( \beta, y_m, m = \text{lqf.}\beta, \text{lqf.y}_m, \text{lqf.m} \)

    N = length(a_hist) - 1
    W, W_m = construct_W_and_Wm(lqf, N)
```
\[ F = \text{lufact}(W, \text{Val}\{\text{true}\}) \]
\[ L, U = F[:L], F[:U] \]
\[ D = \text{diagm}(1.0./\text{diag}(U)) \]
\[ U = D * U \]
\[ L = L * \text{diagm}(1.0./\text{diag}(D)) \]
\[ J = \text{flipdim}(\text{eye}(N + 1), 2) \]

\begin{verbatim}
if t == nothing  # if the problem is deterministic
    a_hist = J * a_hist
    #--------------------------------------------
    # Transform the a sequence if \( \beta \) is given
    #--------------------------------------------
    if \( \beta \) != 1
        a_hist = reshape(a_hist * (\( \beta \)^(collect(N:0)/2)), N + 1, 1)
    end
    a_bar = a_hist - W_m * y_m  # a_bar from the lecture
    Uy = \( L, a_bar \)  # U @ y_bar = \( L^{-1}a_bar \) from the lecture
    y_bar = \( U, Uy \)  # y_bar = \( U^{-1}L^{-1}a_bar \)
    J = flipdim(eye(N + m + 1), 2)
    y_hist = J * vcat(y_bar, y_m)  # y_hist : concatenated y_m and y
    #--------------------------------------------
    # Transform the optimal sequence back if \( \beta \) is given
    #--------------------------------------------
    if \( \beta \) != 1
        y_hist = y_hist .* \( \beta \)^(collect(-m:N)/2)
    end
else  # if the problem is stochastic and we look at it
    Ea_hist = reshape(predict(a_hist, t), N + 1, 1)
    Ea_hist = J * Ea_hist
    a_bar = Ea_hist - W_m * y_m  # a_bar from the lecture
    Uy = \( L, a_bar \)  # U @ y_bar = \( L^{-1}a_bar \) from the lecture
    y_bar = \( U, Uy \)  # y_bar = \( U^{-1}L^{-1}a_bar \)
    J = flipdim(eye(N + m + 1), 2)
    y_hist = J * vcat(y_bar, y_m)  # y_hist : concatenated y_m and y
    end
end
return y_hist, L, U, y_bar
end
\end{verbatim}
QuantEcon.lectures-julia PDF, Release 2018-Aug-8

Example
In this application well have one lag, with
d(L)yt = γ(I − L)yt = γ(yt − yt−1 )
Suppose for the moment that γ = 0
Then the intertemporal component of the LQ problem disappears, and the agent simply wants to maximize
at yt − hyt2 /2 in each period
This means that the agent chooses yt = at /h
In the following well set h = 1, so that the agent just wants to track the {at } process
However, as we increase γ, the agent gives greater weight to a smooth time path
Hence {yt } evolves as a smoothed version of {at }
The {at } sequence well choose as a stationary cyclic process plus some white noise
Heres some code that generates a plot when γ = 0.8
using PyPlot
# == Set seed and generate a_t sequence == #
srand(123)
n = 100
a_seq = sin.(linspace(0, 5 * pi, n)) + 2 + 0.1 * randn(n)
function plot_simulation(;γ=0.8, m=1, h=1., y_m=2.)
d = γ * [1, -1]
y_m = [y_m]
testlq = LQFilter(d, h, y_m)
y_hist, L, U, y = optimal_y(testlq, a_seq)
y = y[end:-1:1] # reverse y
# == Plot simulation results == #
fig, ax = subplots(figsize=(10, 6))
ax[:set_xlabel]("Time")
# == Some fancy plotting stuff -- simplify if you prefer == #
bbox = (0., 1.01, 1., .101)
time = 1:length(y)
ax[:set_xlim](0, maximum(time))
ax[:plot](time, a_seq / h, "k-o", ms=4, lw=2, alpha=0.6, label=L"$a_t$")
ax[:plot](time, y, "b-o", ms=4, lw=2, alpha=0.6, label=L"$y_t$")
ax[:grid]()
ax[:legend](ncol=2, bbox_to_anchor= bbox, loc=3, mode = "expand", fontsize=
,→16)
end
plot_simulation()

5.5. Classical Control with Linear Algebra

865


Here's what happens when we change $\gamma$ to 5.0

```julia
plot_simulation(γ=5.0)
```
And here $\gamma = 10$

```julia
plot_simulation(γ=10.0)
```
5.5.7 Exercises

Exercise 1

Consider solving a discounted version \((\beta < 1)\) of problem \((5.27)\), as follows

Convert \((5.27)\) to the undiscounted problem \((5.48)\)

Let the solution of \((5.48)\) in feedback form be

\[
(1 - \tilde{\lambda}_1 L) \cdots (1 - \tilde{\lambda}_m L) \hat{y}_t = \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k}
\]

or

\[
\hat{y}_t = \tilde{f}_1 \hat{y}_{t-1} + \cdots + \tilde{f}_m \hat{y}_{t-m} + \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k}
\]

Here

- \(h + \tilde{d}(z^{-1})\tilde{d}(z) = \tilde{c}(z^{-1})\tilde{c}(z)\)
- \(\tilde{c}(z) = [(-1)^m \tilde{z}_0 \tilde{z}_1 \cdots \tilde{z}_m]^{1/2} (1 - \tilde{\lambda}_1 z) \cdots (1 - \tilde{\lambda}_m z)\)
where the \( \tilde{z}_j \) are the zeros of \( h + \tilde{d}(z^{-1}) \tilde{d}(z) \).

Prove that (5.51) implies that the solution for \( y_t \) in feedback form is

\[
y_t = f_1 y_{t-1} + \ldots + f_m y_{t-m} + \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} \beta^k \lambda_j^k a_{t+k}
\]

where \( f_j = \tilde{f}_j \beta^{-j/2} \), \( A_j = \tilde{A}_j \), and \( \lambda_j = \tilde{\lambda}_j \beta^{-1/2} \).

**Exercise 2**

Solve the optimal control problem, maximize

\[
\sum_{t=0}^{2} \left\{ a_t y_t - \frac{1}{2} [(1 - 2L) y_t]^2 \right\}
\]

subject to \( y_{-1} \) given, and \( \{a_t\} \) a known bounded sequence.

Express the solution in the feedback form (5.46), giving numerical values for the coefficients.

Make sure that the boundary conditions (5.31) are satisfied.

(Note: this problem differs from the problem in the text in one important way: instead of \( h > 0 \) in (5.27), \( h = 0 \). This has an important influence on the solution.)

**Exercise 3**

Solve the infinite time optimal control problem to maximize

\[
\lim_{N \to \infty} \sum_{t=0}^{N} \left[ a_t y_t - \frac{1}{2} [(1 - 2L) y_t]^2 \right],
\]

subject to \( y_{-1} \) given. Prove that the solution is

\[
y_t = 2y_{t-1} = 2^{t+1} y_{-1} \quad t > 0
\]

**Exercise 4**

Solve the infinite time problem, to maximize

\[
\lim_{N \to \infty} \sum_{t=0}^{N} \left( .0000001 \right) y_t^2 - \frac{1}{2} [(1 - 2L) y_t]^2
\]

subject to \( y_{-1} \) given. Prove that the solution \( y_t = 2y_{t-1} \) violates condition (5.38), and so is not optimal.

Prove that the optimal solution is approximately \( y_t = .5y_{t-1} \).
5.6 Classical Filtering With Linear Algebra

5.6.1 Overview

This is a sequel to the earlier lecture *Classical Control with Linear Algebra*. That lecture used linear algebra – in particular, the LU decomposition – to formulate and solve a class of linear-quadratic optimal control problems.

In this lecture, we will be using a closely related decomposition, the Cholesky decomposition, to solve linear prediction and filtering problems.

We exploit the useful fact that there is an intimate connection between two superficially different classes of problems:

- deterministic linear-quadratic (LQ) optimal control problems
- linear least squares prediction and filtering problems

The first class of problems involves no randomness, while the second is all about randomness.

Nevertheless, essentially the same mathematics solves both type of problem.

This connection, which is often termed duality, is present whether one uses classical or recursive solution procedures.

In fact we saw duality at work earlier when we formulated control and prediction problems recursively in lectures *LQ dynamic programming problems, A first look at the Kalman filter*, and *The permanent income model*.

A useful consequence of duality is that

- With every LQ control problem there is implicitly affiliated a linear least squares prediction or filtering problem.
- With every linear least squares prediction or filtering problem there is implicitly affiliated a LQ control problem.

An understanding of these connections has repeatedly proved useful in cracking interesting applied problems.
For example, Sargent [Sar87] [chs. IX, XIV] and Hansen and Sargent [HS80] formulated and solved control and filtering problems using \( z \)-transform methods.

In this lecture we investigate these ideas using mostly elementary linear algebra.

References

Useful references include [Whi63], [HS80], [Orf88], [AP91], and [Mut60].

5.6.2 Infinite Horizon Prediction and Filtering Problems

We pose two related prediction and filtering problems.

We let \( Y_t \) be a univariate \( m \)th order moving average, covariance stationary stochastic process,

\[
Y_t = d(L)u_t
\]

(5.52)

where \( d(L) = \sum_{j=0}^{m} d_j L^j \), and \( u_t \) is a serially uncorrelated stationary random process satisfying

\[
\begin{align*}
\mathbb{E}u_t &= 0 \\
\mathbb{E}u_t u_s &= \begin{cases} 1 & \text{if } t = s \\ 0 & \text{otherwise} \end{cases}
\end{align*}
\]

(5.53)

We impose no conditions on the zeros of \( d(z) \).

A second covariance stationary process is \( X_t \) given by

\[
X_t = Y_t + \varepsilon_t
\]

(5.54)

where \( \varepsilon_t \) is a serially uncorrelated stationary random process with \( \mathbb{E}\varepsilon_t = 0 \) and \( \mathbb{E}\varepsilon_t \varepsilon_s = 0 \) for all distinct \( t \) and \( s \).

We also assume that \( \mathbb{E}\varepsilon_t \varepsilon_s = 0 \) for all \( t \) and \( s \).

The linear least squares prediction problem is to find the \( L_2 \) random variable \( \hat{X}_{t+j} \) among linear combinations of \( \{X_t, X_{t-1}, \ldots\} \) that minimizes \( \mathbb{E}(\hat{X}_{t+j} - X_{t+j})^2 \).

That is, the problem is to find a \( \gamma_j(L) = \sum_{k=0}^{\infty} \gamma_{jk} L^k \) such that \( \sum_{k=0}^{\infty} |\gamma_{jk}|^2 < \infty \) and \( \mathbb{E}[\gamma_j(L)X_t - X_{t+j}]^2 \) is minimized.

The linear least squares filtering problem is to find a \( b(L) = \sum_{j=0}^{\infty} b_j L^j \) such that \( \sum_{j=0}^{\infty} |b_j|^2 < \infty \) and \( \mathbb{E}[b(L)X_t - Y_t]^2 \) is minimized.

Interesting versions of these problems related to the permanent income theory were studied by [Mut60].

5.6. Classical Filtering With Linear Algebra
Problem formulation

These problems are solved as follows

The covariograms of $Y$ and $X$ and their cross covariogram are, respectively,

$$
C_X(\tau) = E X_t X_{t-\tau} \\
C_Y(\tau) = E Y_t Y_{t-\tau} \quad \tau = 0, \pm 1, \pm 2, \ldots \\
C_{Y,X}(\tau) = E Y_t X_{t-\tau}
$$

The covariance and cross covariance generating functions are defined as

$$
g_X(z) = \sum_{\tau=-\infty}^{\infty} C_X(\tau) z^\tau \\
g_Y(z) = \sum_{\tau=-\infty}^{\infty} C_Y(\tau) z^\tau \\
g_{Y,X}(z) = \sum_{\tau=-\infty}^{\infty} C_{Y,X}(\tau) z^\tau
$$

The generating functions can be computed by using the following facts

Let $v_{1t}$ and $v_{2t}$ be two mutually and serially uncorrelated white noises with unit variances

That is, $E v_{1t}^2 = E v_{2t}^2 = 1$, $E v_{1t} = E v_{2t} = 0$, $E v_{1t} v_{2s} = 0$ for all $t$ and $s$, $E v_{1t} v_{1t-j} = E v_{2t} v_{2t-j} = 0$ for all $j \neq 0$

Let $y_t$ and $x_t$ be two random process given by

$$
y_t = A(L) v_{1t} + B(L) v_{2t} \\
x_t = C(L) v_{1t} + D(L) v_{2t}
$$

Then, as shown for example in [Sar87] [ch. XII], it is true that

$$
g_y(z) = A(z) A(z^{-1}) + B(z) B(z^{-1}) \\
g_x(z) = C(z) C(z^{-1}) + D(z) D(z^{-1}) \\
g_{y,x}(z) = A(z) C(z^{-1}) + B(z) D(z^{-1})
$$

Applying these formulas to (5.52) – (5.55), we have

$$
g_Y(z) = d(z) d(z^{-1}) \\
g_X(z) = d(z) d(z^{-1}) + h \\
g_{Y,X}(z) = d(z) d(z^{-1})
$$

The key step in obtaining solutions to our problems is to factor the covariance generating function $g_X(z)$ of $X$. 

Chapter 5. Time Series Models
The solutions of our problems are given by formulas due to Wiener and Kolmogorov.
These formulas utilize the Wold moving average representation of the $X_t$ process,

$$X_t = c(L) \eta_t$$  \hspace{1cm} (5.59)

where $c(L) = \sum_{j=0}^{m} c_j L^j$, with

$$c_0 \eta_t = X_t - \hat{E}[X_t|X_{t-1}, X_{t-2}, \ldots]$$  \hspace{1cm} (5.60)

Here $\hat{E}$ is the linear least squares projection operator.

Equation (5.60) is the condition that $c_0 \eta_t$ can be the one-step ahead error in predicting $X_t$ from its own past values.

Condition (5.60) requires that $\eta_t$ lie in the closed linear space spanned by $[X_t, X_{t-1}, \ldots]$
This will be true if and only if the zeros of $c(z)$ do not lie inside the unit circle.

It is an implication of (5.60) that $\eta_t$ is a serially uncorrelated random process, and that a normalization can be imposed so that $\mathbb{E}\eta_t^2 = 1$

Consequently, an implication of (5.59) is that the covariance generating function of $X_t$ can be expressed as

$$g_X(z) = c(z) c(z^{-1})$$  \hspace{1cm} (5.61)

It remains to discuss how $c(L)$ is to be computed.

Combining (5.57) and (5.61) gives

$$d(z) d(z^{-1}) + h = c(z) c(z^{-1})$$  \hspace{1cm} (5.62)

Therefore, we have already showed constructively how to factor the covariance generating function $g_X(z) = d(z) d(z^{-1}) + h$

We now introduce the **annihilation operator**: 

$$\left[ \sum_{j=-\infty}^{\infty} f_j L^j \right]_+ \equiv \sum_{j=0}^{\infty} f_j L^j$$  \hspace{1cm} (5.63)

In words, $[ \ ]_+$ means ignore negative powers of $L$.

We have defined the solution of the prediction problem as $\hat{E}[X_{t+j}|X_t, X_{t-1}, \ldots] = \gamma_j(L)X_t$

Assuming that the roots of $c(z) = 0$ all lie outside the unit circle, the Wiener-Kolmogorov formula for $\gamma_j(L)$ holds:
\[
\gamma_j (L) = \left[ \frac{c(L)}{L^j} \right] + c(L)^{-1}
\]  \hfill (5.64)

We have defined the solution of the filtering problem as \( \hat{E}[Y_t \mid X_t, X_{t-1}, \ldots] = b(L)X_t \)

The Wiener-Kolomogorov formula for \( b(L) \) is

\[
b(L) = \left( \frac{gyx(L)}{c(L^{-1})} \right) + c(L)^{-1}
\]

or

\[
b(L) = \left[ \frac{d(L)d(L^{-1})}{c(L^{-1})} \right] + c(L)^{-1}
\]  \hfill (5.65)

Formulas (5.64) and (5.65) are discussed in detail in \cite{Whi83} and \cite{Sar87}

The interested reader can there find several examples of the use of these formulas in economics Some classic examples using these formulas are due to \cite{Mut60}

As an example of the usefulness of formula (5.65), we let \( X_t \) be a stochastic process with Wold moving average representation

\[
X_t = c(L)\eta_t
\]

where \( \mathbb{E}\eta_t^2 = 1 \), and \( c_0\eta_t = X_t - \hat{E}[X_t|X_{t-1}, \ldots], c(L) = \sum_{j=0}^{m} c_j L^j \)

Suppose that at time \( t \), we wish to predict a geometric sum of future \( X \) s, namely

\[
y_t \equiv \sum_{j=0}^{\infty} \delta^j X_{t+j} = \frac{1}{1 - \delta L^{-1}}X_t
\]

given knowledge of \( X_t, X_{t-1}, \ldots \)

We shall use (5.65) to obtain the answer

Using the standard formulas (5.57), we have that

\[
g_{yx}(z) = (1 - \delta z^{-1})c(z)c(z^{-1})
\]

\[
g_x(z) = c(z)c(z^{-1})
\]

Then (5.65) becomes

\[
b(L) = \left[ \frac{c(L)}{1 - \delta L^{-1}} \right] + c(L)^{-1}
\]  \hfill (5.66)

In order to evaluate the term in the annihilation operator, we use the following result from \cite{HS80}

**Proposition**
\[ g(z) = \sum_{j=0}^{\infty} g_j z^j \text{ where } \sum_{j=0}^{\infty} |g_j|^2 < +\infty \]

\[ h(z^{-1}) = (1 - \delta_1 z^{-1}) \cdots (1 - \delta_n z^{-1}) \text{ where } |\delta_j| < 1, \text{ for } j = 1, \ldots, n \]

Then

\[
\left[ \frac{g(z)}{h(z^{-1})} \right]_+ = \frac{g(z)}{h(z^{-1})} - \sum_{j=1}^{n} \frac{\delta_j g(\delta_j)}{\prod_{k=1}^{n} (\delta_j - \delta_k)} \left( \frac{1}{z - \delta_j} \right)
\]

(5.67)

and, alternatively,

\[
\left[ \frac{g(z)}{h(z^{-1})} \right]_+ = \sum_{j=1}^{n} B_j \left( \frac{z g(z) - \delta_j g(\delta_j)}{z - \delta_j} \right)
\]

(5.68)

where \( B_j = 1/ \prod_{k=1}^{n} (1 - \delta_k/\delta_j) \)

Applying formula (5.68) of the proposition to evaluating (5.66) with \( g(z) = c(z) \) and \( h(z^{-1}) = 1 - \delta z^{-1} \) gives

\[ b(L) = \left[ \frac{L c(L) - \delta c(\delta)}{L - \delta} \right] c(L)^{-1} \]

or

\[ b(L) = \left[ \frac{1 - \delta c(\delta) L^{-1} c(L)^{-1}}{1 - \delta L^{-1}} \right] \]

Thus, we have

\[
\mathbb{E} \left[ \sum_{j=0}^{\infty} \delta^j X_{t+j} | X_t, x_{t-1}, \ldots \right] = \left[ \frac{1 - \delta c(\delta) L^{-1} c(L)^{-1}}{1 - \delta L^{-1}} \right] X_t
\]

(5.69)

This formula is useful in solving stochastic versions of problem 1 of lecture Classical Control with Linear Algebra in which the randomness emerges because \( \{a_t\} \) is a stochastic process.

The problem is to maximize

\[
\mathbb{E}_0 \lim_{N \to \infty} \sum_{t=0}^{N} \beta^t \left[ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} \left[ d(L)y_t \right]^2 \right]
\]

(5.70)

where \( \mathbb{E}_t \) is mathematical expectation conditioned on information known at \( t \), and where \( \{a_t\} \) is a covariance stationary stochastic process with Wold moving average representation

\[ a_t = c(L) \eta_t \]
where

\[ c(L) = \sum_{j=0}^{\hat{n}} c_j L^j \]

and

\[ \eta_t = a_t - \mathbb{E}[a_t|a_{t-1}, \ldots] \]

The problem is to maximize (5.70) with respect to a contingency plan expressing \( y_t \) as a function of information known at \( t \), which is assumed to be \( (y_{t-1}, y_{t-2}, \ldots, a_t, a_{t-1}, \ldots) \).

The solution of this problem can be achieved in two steps:

First, ignoring the uncertainty, we can solve the problem assuming that \( \{a_t\} \) is a known sequence.

The solution is, from above,

\[ c(L)y_t = c(\beta L^{-1})a_t \]

or

\[ (1 - \lambda_1 L) \ldots (1 - \lambda_m L)y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} (\lambda_j \beta)^k a_{t+k} \quad (5.71) \]

Second, the solution of the problem under uncertainty is obtained by replacing the terms on the right-hand side of the above expressions with their linear least squares predictors.

Using (5.69) and (5.71), we have the following solution:

\[ (1 - \lambda_1 L) \ldots (1 - \lambda_m L)y_t = \sum_{j=1}^{m} A_j \left[ \frac{1 - \beta \lambda_j c(\beta \lambda_j) L^{-1} c(L)^{-1}}{1 - \beta \lambda_j L^{-1}} \right] a_t \]

### 5.6.3 Finite Dimensional Prediction

Let \( (x_1, x_2, \ldots, x_T)' = x \) be a \( T \times 1 \) vector of random variables with mean \( \mathbb{E}x = 0 \) and covariance matrix \( \mathbb{E}xx' = V \).

Here \( V \) is a \( T \times T \) positive definite matrix.

We shall regard the random variables as being ordered in time, so that \( x_t \) is thought of as the value of some economic variable at time \( t \).

For example, \( x_t \) could be generated by the random process described by the Wold representation presented in equation (5.59).

In this case, \( V_{ij} \) is given by the coefficient on \( z^{i-j} \) in the expansion of \( g_x(z) = d(z) d(z^{-1}) + h \), which equals \( h + \sum_{k=0}^{\infty} d_k d_{k+i-j} \).

We shall be interested in constructing \( j \) step ahead linear least squares predictors of the form

\[ \mathbb{E} [x_T|x_{T-j}, x_{T-j+1}, \ldots, x_1] \]
where $\mathbb{E}$ is the linear least squares projection operator

The solution of this problem can be exhibited by first constructing an orthonormal basis of random variables $\varepsilon$ for $x$

Since $V$ is a positive definite and symmetric, we know that there exists a (Cholesky) decomposition of $V$ such that

$$V = L^{-1}(L^{-1})'$$

or

$$LV L' = I$$

where $L$ is lower-trangular, and therefore so is $L^{-1}$

Form the random variable $Lx = \varepsilon$

Then $\varepsilon$ is an orthonormal basis for $x$, since $L$ is nonsingular, and $\mathbb{E}\varepsilon\varepsilon' = L\mathbb{E}xx'L' = I$

It is convenient to write out the equations $Lx = \varepsilon$ and $L^{-1}\varepsilon = x$

$$L_{11}x_1 = \varepsilon_1$$
$$L_{21}x_1 + L_{22}x_2 = \varepsilon_2$$
$$\vdots$$
$$L_{T1}x_1 \ldots + L_{TT}x_T = \varepsilon_T$$

(5.72)

or

$$\sum_{j=0}^{t-1} L_{t,t-j}x_{t-j} = \varepsilon_t, \quad t = 1, 2, \ldots T$$

(5.73)

We also have

$$x_t = \sum_{j=0}^{t-1} L_{t,t-j}^{-1} \varepsilon_{t-j}.$$  \hspace{1cm} (5.74)

Notice from (5.74) that $x_t$ is in the space spanned by $\varepsilon_t, \varepsilon_{t-1}, \ldots, \varepsilon_1$, and from (5.73) that $\varepsilon_t$ is in the space spanned by $x_t, x_{t-1}, \ldots, x_1$

Therefore, we have that for $t - 1 \geq m \geq 1$

$$\mathbb{E}[x_t \mid x_{t-m}, x_{t-m-1}, \ldots, x_1] = \mathbb{E}[x_t \mid \varepsilon_{t-m}, \varepsilon_{t-m-1}, \ldots, \varepsilon_1]$$

(5.75)

For $t - 1 \geq m \geq 1$ rewrite (5.74) as
\[ x_t = \sum_{j=0}^{m-1} L_{t,t-j}^{-1} \varepsilon_{t-j} + \sum_{j=m}^{t-1} L_{t,t-j}^{-1} \varepsilon_{t-j} \] (5.76)

Representation (5.76) is an orthogonal decomposition of \( x_t \) into a part \( \sum_{j=0}^{t-1} L_{t,t-j}^{-1} \varepsilon_{t-j} \) that lies in the space spanned by \( [x_{t-m}, x_{t-m+1}, \ldots, x_1] \), and an orthogonal component not in this space.

**Implementation**

Code that computes solutions to LQ control and filtering problems using the methods described here and in *Classical Control with Linear Algebra* can be found in the file `control_and_filter.jl`.

Here’s how it looks:

```julia
#=
Author: Shunsuke Hori
=#

using Polynomials

struct LQFilter{TR<:Real, TI<:Integer, TF<:AbstractFloat}
  d::Vector{TF}
  h::TR
  y_m::Vector{TF}
  m::TI
  ::Vector{TF}
  β::TR
  _r::Union{Vector{TF},Void}
  k::Union(TI,Void)
end

"""
Parameters
--------
d : Vector
  The order of the coefficients: \([d_0, d_1, \ldots, d_m]\)
h : Real
  Parameter of the objective function (corresponding to the quadratic term)
y_m : Vector
  Initial conditions for \( y \)
r : Vector
  The order of the coefficients: \([r_0, r_1, \ldots, r_k]\) (optional, if not defined -> deterministic problem)
β : Real or nothing
  Discount factor (optional, default value is one)
h_eps :
  ""
```

---

878 Chapter 5. Time Series Models
function LQFilter{TR<:Real}(d::Vector{TR},
    h::TR,
    y_m::Vector{TR};
    r::Union{Vector{TR},Void}=nothing,
    β::Union{TR,Void}=nothing,
    h_eps::Union{TR,Void}=nothing,
)

    m = length(d) - 1
    m == length(y_m) ||
        throw(ArgumentError("y_m and d must be of same length = $m"))

    # Define the coefficients of up front
    #---------------------------------------------
    = Vector{TR}(2m + 1)
    for i in -m:m
        [m-i+1] = sum(diag(d*d', -i))
    end
    [m+1] = [m+1] + h

    # If r is given calculate the vector _r
    #-----------------------------------------------------
    if r == nothing
        k=nothing
        _r = nothing
    else
        k = size(r, 1) - 1
        _r = Vector{TR}(2k + 1)
        for i = -k:k
            _r[k-i+1] = sum(diag(r*r', -i))
        end

        if h_eps != nothing
            _r[k+1] = _r[k+1] + h_eps
        end
    end

    # If β is given, define the transformed variables
    #-----------------------------------------------------
    if β == nothing
        β = 1.0
    else
        d = β.(collect(0:m)/2) * d
        y_m = y_m * β.^(collect(1:m)/2)
    end

5.6. Classical Filtering With Linear Algebra
function construct_W_and_Wm(lqf::LQFilter, N::Integer)
    d, m = lqf.d, lqf.m
    W = zeros(N + 1, N + 1)
    W_m = zeros(N + 1, m)

    # Terminal conditions
    D_m1 = zeros(m + 1, m + 1)
    M = zeros(m + 1, m)

    # (1) Construct the D_{m+1} matrix using the formula
    for j in 1:(m+1)
        for k in j:(m+1)
            D_m1[j, k] = dot(d[1:j, 1], d[k-j+1:k, 1])
        end
    end

    # Make the matrix symmetric
    D_m1 = D_m1 + D_m1' - diagm(diag(D_m1))

    # (2) Construct the M matrix using the entries of D_m1
    for j in 1:m
        for i in (j + 1):(m + 1)
            M[i, j] = D_m1[i-j, m+1]
        end
    end

    M

    # Euler equations for t = 0, 1, ..., N-(m+1)
    for (i, row) in enumerate((m + 2):(N + 1 - m))
        W[row, (i + 1):(2m + 1 + i)] = '
    end

    return LQFilter(d, h, y_m, m, , β, _r, k)
end

""
This constructs the matrices W and W_m for a given number of periods N
""

Chapter 5. Time Series Models

880

QuantEcon.lectures-julia PDF, Release 2018-Aug-8
for i in 1:m
    W[N - m + i + 1 , end=(2m + 1 - i)+1:end] = [1:end-i]
end

for i in 1:m
    W_m[N - i + 2, 1:(m - i)+1] = [(m + 1 + i):end]
end

return W, W_m
end

""
This function calculates z_0 and the 2m roots of the characteristic equation
associated with the Euler equation (1.7)
Note:
      ______
  `poly(roots)` from Polynomial.jll defines a polynomial using its roots that
  can be evaluated at any point by `polyval(Poly,x)`.
If x_1, x_2, ... , x_m are the roots then
  polyval(poly(roots),x) = (x - x_1)(x - x_2)...(x - x_m)
""

function roots_of_characteristic(lqf::LQFilter)
    m, = lqf.m, lqf.

    # Calculate the roots of the 2m-polynomial
    _poly=Poly([end:-1:1])
    proots = roots(_poly)

    # sort the roots according to their length (in descending order)
    roots_sorted = sort(proots, by=abs)[end:-1:1]

    z_0 = sum() / polyval(poly(proots), 1.0)
    z_1_to_m = roots_sorted[1:m]  # we need only those outside the unit circle

    λ = 1 ./ z_1_to_m
    return z_1_to_m, z_0, λ
end

""
This function computes the coefficients \(c_j\), \(j = 0, 1, \ldots, m\) for
\(c(z) = \sum_{j=0}^{m} c_j z^j\)
Based on the expression (1.9). The order is
\(c\_coeffs = [c_0, c_1, \ldots, c_{(m-1)}, c_m]\)
""

function coeffs_of_c(lqf::LQFilter)
    m = lqf.m
    z_1_to_m, z_0, λ = roots_of_characteristic(lqf)

    c_0 = (z_0 * prod(z_1_to_m) * (-1.0)^m)^0.5
    c_coeffs = coeffs(poly(z_1_to_m)) * z_0 / c_0

    return c_coeffs
end

"""
This function calculates \( \{\lambda_j, j=1,\ldots,m\} \) and \( \{A_j, j=1,\ldots,m\} \)
of the expression (1.15)

```
function solution(lqf::LQFilter)
    z_1_to_m, z_0, \lambda = roots_of_characteristic(lqf)
    c_0 = coeffs_of_c(lqf)[end]
    A = zeros(lqf.m)
    for j in 1:m
        denom = 1 - \lambda/\lambda[j]
        A[j] = c_0^(-2) / prod(denom[1:m .!= j])
    end
    return \lambda, A
end
```

This function constructs the covariance matrix for \( x^N \) (see section 6) for a given period \( N \)

```
function construct_V(lqf::LQFilter; N::Integer=nothing)
    if N == nothing
        error("N must be provided!!")
    end
    if !(typeof(N) <: Integer)
        throw(ArgumentError("N must be Integer!"))
    end

    _r, k = lqf._r, lqf.k
    V = zeros(N, N)
    for i in 1:N
        for j in 1:N
            if abs(i-j) <= k
                V[i, j] = _r[k + abs(i-j)+1]
            end
        end
    end
    return V
end
```

Assuming that the \( u \)'s are normal, this method draws a random path for \( x^N \)

```
function simulate_a(lqf::LQFilter, N::Integer)
    V = construct_V(N + 1)
    d = MVNSampler(zeros(N + 1), V)
    return rand(d)
end
```

This function implements the prediction formula discussed in section 6 (1.59) it takes a realization
for \( a^N \), and the period in which the prediction is
function predict(lqf::LQFilter, a_hist::Vector, t::Integer)
    N = length(a_hist) - 1
    V = construct_V(N + 1)

    aux_matrix = zeros(N + 1, N + 1)
    aux_matrix[1:t+1, 1:t+1] = eye(t + 1)
    L = chol(V)'
    Ea_hist = inv(L) * aux_matrix * L * a_hist

    return Ea_hist
end

# if t is NOT given it takes a_hist (Vector or Array) as a deterministic a_t
# if t is given, it solves the combined control prediction problem (section 7)
# (by default, t == nothing -> deterministic)

for a given sequence of a_t (either deterministic or a particular realization),
it calculates the optimal y_t sequence using the method of the lecture

Note:
--------
lufact normalizes L, U so that L has unit diagonal elements
To make things consistent with the lecture, we need an auxiliary diagonal
matrix D which renormalizes L and U

function optimal_y(lqf::LQFilter, a_hist::Vector, t = nothing)
    β, y_m, m = lqf.β, lqf.y_m, lqf.m

    N = length(a_hist) - 1
    W, W_m = construct_W_and_Wm(lqf, N)

    F = lufact(W, Val(true))

    L, U = F[:L], F[:U]
    D = diagm(1.0./diag(U))
    U = D * U
    L = L * diagm(1.0./diag(D))

    J = flipdim(eye(N + 1), 2)

    if t == nothing # if the problem is deterministic
        a_hist = J * a_hist
    else
        a_hist = reshape(a_hist * (β^((collect(N:0)/ 2))), N + 1, 1)
Lets use this code to tackle two interesting examples

**Example 1**

Consider a stochastic process with moving average representation

\[ x_t = (1 - 2L)\varepsilon_t \]

where \( \varepsilon_t \) is a serially uncorrelated random process with mean zero and variance unity.

We want to use the Wiener-Kolmogorov formula (5.64) to compute the linear least squares forecasts \( \mathbb{E}[x_{t+j} | x_t, x_{t-1}, \ldots] \), for \( j = 1, 2 \).

We can do everything we want by setting \( d = r \), generating an instance of \( LQFilter \), then invoking pertinent methods of \( LQFilter \).
The Wold representation is computed by `example.coefficients_of_c()`

Let's check that it flips roots as required

```julia
coeffs_of_c(example)
```

```
2-element Array{Float64,1}:
  2.0
  -1.0
```

```julia
roots_of_characteristic(example)
```

```
([2.0],-2.0,[0.5])
```

Now let's form the covariance matrix of a time series vector of length \( N \) and put it in \( V \)

Then we'll take a Cholesky decomposition of \( V = L^{-1}L^{-1} = L_iL'_i \) and use it to form the vector of moving average representations \( x = L_i\varepsilon \) and the vector of autoregressive representations \( Lx = \varepsilon \)

```julia
V = construct_V(example,N=5)
```

```
5x5 Array{Float64,2}:
  5.0  -2.0  0.0  0.0  0.0
-2.0   5.0  -2.0  0.0  0.0
  0.0  -2.0   5.0  -2.0  0.0
  0.0   0.0  -2.0   5.0  -2.0
  0.0   0.0   0.0  -2.0   5.0
```

Notice how the lower rows of the moving average representations are converging to the appropriate infinite history Wold representation

```julia
F = cholfact(V)
Li = F[:,L]
```

```
5x5 LowerTriangular{Float64,Array{Float64,2}}:
  2.23607
-0.894427  2.04939
  0.0  -0.9759  2.01187
  0.0   0.0  -0.9941  2.00294
  0.0   0.0   0.0  -0.998533  2.00073
```

Notice how the lower rows of the autoregressive representations are converging to the appropriate infinite history autoregressive representation

5.6. Classical Filtering With Linear Algebra
\[ L = \text{inv}(L_i) \]

5.65 LowerTriangular{Float64,Array{Float64,2}}:
0.447214
0.195180 0.487950
0.0946760 0.236691 0.497050
0.04698980 0.117474 0.246696 0.499266
0.02345180 0.0586295 0.123122 0.249176 0.499817

Remark
Let \( \pi(z) = \sum_{j=0}^{m} \pi_j z^j \) and let \( z_1, \ldots, z_k \) be the zeros of \( \pi(z) \) that are inside the unit circle, \( k < m \). Then define
\[
\theta(z) = \pi(z) \left( \frac{(z_1 z - 1)}{(z - z_1)} \right) \left( \frac{(z_2 z - 1)}{(z - z_2)} \right) \ldots \left( \frac{(z_k z - 1)}{(z - z_k)} \right)
\]
The term multiplying \( \pi(z) \) is termed a Blaschke factor.
Then it can be proved directly that
\[
\theta(z^{-1})\theta(z) = \pi(z^{-1})\pi(z)
\]
and that the zeros of \( \theta(z) \) are not inside the unit circle.

Example 2
Consider a stochastic process \( X_t \) with moving average representation
\[
X_t = (1 - \sqrt{2}L^2)\varepsilon_t
\]
where \( \varepsilon_t \) is a serially uncorrelated random process with mean zero and variance unity.
Let's find a Wold moving average representation for \( x_t \).
Let's use the Wiener-Kolomogorov formula (5.64) to compute the linear least squares forecasts \( \hat{E} [X_{t+j} | X_{t-1}, \ldots] \) for \( j = 1, 2, 3 \).
We proceed in the same way as example 1.

\[
m = 2
y_m = [0.0, 0.0]
d = [1, 0, -\sqrt{2}]
r = [1, 0, -\sqrt{2}]
h = 0.0
example = LQFilter(d, h, y_m, r = d)
\]

coeffs_of_c(example)

3-element Array{Float64,1}:
1.41421
-0.0
-1.0
Prediction

It immediately follows from the orthogonality principle of least squares (see [AP91] or [Sar87] ch. X) that

\[
\hat{E}[x_t \mid x_{t-m}, x_{t-m+1}, \ldots x_1] = \sum_{j=m}^{t-1} L_{t,t-j}^{-1} e_{t-j}
\]

(5.77)

\[
= [L_{t,1}^{-1} L_{t,2}^{-1}, \ldots, L_{t,t-m}^{-1} 0 0 \ldots 0] L x
\]
This can be interpreted as a finite-dimensional version of the Wiener-Kolmogorov \( m \)-step ahead prediction formula.

We can use (5.77) to represent the linear least squares projection of the vector \( x \) conditioned on the first \( s \) observations \([x_s, x_{s-1} \ldots, x_1]\)

We have

\[
\mathbb{E}[x \mid x_s, x_{s-1}, \ldots, x_1] = L^{-1} \begin{bmatrix} I_s & 0 \\ 0 & 0_{(t-s)} \end{bmatrix} Lx
\]  

(5.78)

This formula will be convenient in representing the solution of control problems under uncertainty.

Equation (5.74) can be recognized as a finite dimensional version of a moving average representation.

Equation (5.73) can be viewed as a finite dimension version of an autoregressive representation.

Notice that even if the \( x_t \) process is covariance stationary, so that \( V \) is such that \( V_{ij} \) depends only on \(|i-j|\), the coefficients in the moving average representation are time-dependent, there being a different moving average for each \( t \).

If \( x_t \) is a covariance stationary process, the last row of \( L^{-1} \) converges to the coefficients in the Wold moving average representation for \( \{x_t\} \) as \( T \to \infty \).

Further, if \( x_t \) is covariance stationary, for fixed \( k \) and \( j > 0 \), \( L^{-1}_{T,T-j} \) converges to \( L^{-1}_{T-k,T-k-j} \) as \( T \to \infty \).

That is, the bottom rows of \( L^{-1} \) converge to each other and to the Wold moving average coefficients as \( T \to \infty \).

This last observation gives one simple and widely-used practical way of forming a finite \( T \) approximation to a Wold moving average representation.

First, form the covariance matrix \( \mathbb{E}xx' = V \), then obtain the Cholesky decomposition \( L^{-1}L^{-1}' \) of \( V \), which can be accomplished quickly on a computer.

The last row of \( L^{-1} \) gives the approximate Wold moving average coefficients.

This method can readily be generalized to multivariate systems.

### 5.6.4 Combined Finite Dimensional Control and Prediction

Consider the finite-dimensional control problem, maximize

\[
\mathbb{E} \sum_{t=0}^{N} \left\{ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} [d(L)y_t]^2 \right\}, \quad h > 0
\]

where \( d(L) = d_0 + d_1 L + \ldots + d_m L^m \), \( L \) is the lag operator, \( \tilde{a} = [a_N, a_{N-1} \ldots, a_1, a_0]' \) a random vector with mean zero and \( \mathbb{E} \tilde{a}\tilde{a}' = V \)

The variables \( y_{-1}, \ldots, y_{-m} \) are given.

Maximization is over choices of \( y_0, y_1 \ldots, y_N \), where \( y_t \) is required to be a linear function of \( \{y_{t-s-1}, t + m - 1 \geq 0; a_{t-s}, t \geq s \geq 0\} \).
We saw in the lecture *Classical Control with Linear Algebra* that the solution of this problem under certainty could be represented in feedback-feedforward form

\[ U \bar{y} = L^{-1} \bar{a} + K \begin{bmatrix} y_{1-n} \\ \vdots \\ y_{1-m} \end{bmatrix} \]

for some \((N + 1) \times m\) matrix \(K\)

Using a version of formula (5.77), we can express \( \hat{E}[a | a_s, a_{s-1}, \ldots, a_0] \) as

\[ \hat{E}[a | a_s, a_{s-1}, \ldots, a_0] = \tilde{U}^{-1} \begin{bmatrix} 0 & 0 \\ 0 & I_{(s+1)} \end{bmatrix} \tilde{U} \bar{a} \]

where \(I_{(s+1)}\) is the \((s + 1) \times (s + 1)\) identity matrix, and \(V = \tilde{U}^{-1} \tilde{V}^{-1}'\), where \(\tilde{U}\) is the upper triangular Cholesky factor of the covariance matrix \(V\)

(We have reversed the time axis in dating the \(a_s\) relative to earlier)

The time axis can be reversed in representation (5.78) by replacing \(L\) with \(L^T\)

The optimal decision rule to use at time \(0 \leq t \leq N\) is then given by the \((N - t + 1)^{th}\) row of

\[ U \bar{y} = L^{-1} \tilde{U}^{-1} \begin{bmatrix} 0 & 0 \\ 0 & I_{(t+1)} \end{bmatrix} \tilde{U} \bar{a} + K \begin{bmatrix} y_{1-n} \\ \vdots \\ y_{1-m} \end{bmatrix} \]

### 5.6.5 Exercises

**Exercise 1**

Let \(Y_t = (1 - 2L)u_t\) where \(u_t\) is a mean zero white noise with \(\mathbb{E}u_t^2 = 1\). Let

\[ X_t = Y_t + \varepsilon_t \]

where \(\varepsilon_t\) is a serially uncorrelated white noise with \(\mathbb{E}\varepsilon_t^2 = 9\), and \(\mathbb{E}\varepsilon_t u_s = 0\) for all \(t\) and \(s\)

Find the Wold moving average representation for \(X_t\)

Find a formula for the \(A_{1j}\)s in

\[ \mathbb{E} \widehat{X}_{t+1} \mid X_t, X_{t-1}, \ldots = \sum_{j=0}^{\infty} A_{1j} X_{t-j} \]

Find a formula for the \(A_{2j}\)s in

\[ \mathbb{E} X_{t+2} \mid X_t, X_{t-1}, \ldots = \sum_{j=0}^{\infty} A_{2j} X_{t-j} \]
Exercise 2

(Multivariable Prediction) Let \( Y_t \) be an \((n \times 1)\) vector stochastic process with moving average representation

\[
Y_t = D(L)U_t
\]

where \( D(L) = \sum_{j=0}^{m} D_j L^j \), \( D_j \) an \( n \times n \) matrix, \( U_t \) an \((n \times 1)\) vector white noise with \( \mathbb{E} U_t = 0 \) for all \( t \), \( \mathbb{E} U_t U_t' = 0 \) for all \( s \neq t \), and \( \mathbb{E} U_t U_t' = I \) for all \( t \).

Let \( \epsilon_t \) be an \( n \times 1 \) vector white noise with mean 0 and contemporaneous covariance matrix \( H \), where \( H \) is a positive definite matrix.

Let \( X_t = Y_t + \epsilon_t \)

Define the covariograms as

\[
C_X(\tau) = \mathbb{E} X_t X_{t-\tau}', C_Y(\tau) = \mathbb{E} Y_t Y_{t-\tau}', C_{YX}(\tau) = \mathbb{E} Y_t X_{t-\tau}
\]

Then define the matrix covariance generating function, as in (5.47), only interpret all the objects in (5.47) as matrices.

Show that the covariance generating functions are given by

\[
g_y(z) = D(z)D(z^{-1})', \quad g_X(z) = D(z)D(z^{-1})' + H, \quad g_{YX}(z) = D(z)D(z^{-1})'
\]

A factorization of \( g_X(z) \) can be found (see [Roz67] or [Whi83]) of the form

\[
D(z)D(z^{-1})' + H = C(z)C(z^{-1})', \quad C(z) = \sum_{j=0}^{m} C_j z^j
\]

where the zeros of \( |C(z)| \) do not lie inside the unit circle.

A vector Wold moving average representation of \( X_t \) is then

\[
X_t = C(L)\eta_t
\]

where \( \eta_t \) is an \((n \times 1)\) vector white noise that is fundamental for \( X_t \).

That is, \( X_t - \mathbb{E} [X_t | X_{t-1}, X_{t-2} \ldots] = C_0 \eta_t \)

The optimum predictor of \( X_{t+j} \) is

\[
\hat{\mathbb{E}} [X_{t+j} | X_t, X_{t-1}, \ldots] = \left[ \frac{C(L)}{L^j} \right]_+ \eta_t
\]

If \( C(L) \) is invertible, i.e., if the zeros of \( \text{det} C(z) \) lie strictly outside the unit circle, then this formula can be written

\[
\hat{\mathbb{E}} [X_{t+j} | X_t, X_{t-1}, \ldots] = \left[ \frac{C(L)}{L^j} \right]_+ C(L)^{-1} X_t
\]
Here we look at models in which a value function for one Bellman equation has as an argument the value function for another Bellman equation.

6.1 Dynamic Stackelberg Problems

Contents

- Dynamic Stackelberg Problems
  - Overview
  - The Stackelberg Problem
  - Solving the Stackelberg Problem
  - Shadow prices
  - A Large Firm With a Competitive Fringe
  - Concluding Remarks
  - Exercises

6.1.1 Overview

Previous lectures including *LQ* dynamic programming, rational expectations equilibrium, and Markov perfect equilibrium lectures have studied decision problems that are recursive in what we can call natural state variables, such as

- stocks of capital (fiscal, financial and human)
- wealth
- information that helps forecast future prices and quantities that impinge on future payoffs

Optimal decision rules are functions of the natural state variables in problems that are recursive in the natural state variables.
In this lecture, we describe problems that are not recursive in the natural state variables. Kydland and Prescott \cite{KP77}, \cite{Pre77} and Calvo \cite{Cal78} gave examples of such decision problems. These problems have the following features:

- Time $t \geq 0$ actions of decision makers called followers depend on time $s \geq t$ decisions of another decision maker called a Stackelberg leader.
- At time $t = 0$, the Stackelberg leader chooses his actions for all times $s \geq 0$.
- In choosing actions for all times at time 0, the Stackelberg leader can be said to commit to a plan.
- The Stackelberg leader has distinct optimal decision rules at time $t = 0$, on the one hand, and at times $t \geq 1$, on the other hand.
- The Stackelberg leader's decision rules for $t = 0$ and $t \geq 1$ have distinct state variables.
- Variables that encode history dependence appear in optimal decision rules of the Stackelberg leader at times $t \geq 1$.
- These properties of the Stackelberg leader's decision rules are symptoms of the time inconsistency of optimal government plans.

An example of a time inconsistent optimal rule is that of a large agent (e.g., a government) that confronts a competitive market composed of many small private agents, and in which private agents' decisions at each date are influenced by their forecasts of the large agent's future actions.

The rational expectations equilibrium concept plays an essential role. A rational expectations restriction implies that when it chooses its future actions, the Stackelberg leader also chooses the followers' expectations about those actions.

The Stackelberg leader understands and exploits that situation. In a rational expectations equilibrium, the Stackelberg leader's time $t$ actions confirm private agents' forecasts of those actions.

The requirement to confirm prior followers' forecasts puts constraints on the Stackelberg leader's time $t$ decisions that prevent its problem from being recursive in natural state variables.

These additional constraints make the Stackelberg leader's decision rule at $t$ depend on the entire history of the natural state variables from time 0 to time $t$.

This lecture displays these principles within the tractable framework of linear quadratic problems.

It is based on chapter 19 of \cite{LS18}.

### 6.1.2 The Stackelberg Problem

We use the optimal linear regulator (a.k.a. the linear-quadratic dynamic programming problem described in \textit{LQ Dynamic Programming problems}) to solve a linear quadratic version of what is known as a dynamic Stackelberg problem.
For now we refer to the Stackelberg leader as the government and the Stackelberg follower as the representative agent or private sector.

Soon well give an application with another interpretation of these two decision makers.

Let \( z_t \) be an \( n_z \times 1 \) vector of natural state variables.

Let \( x_t \) be an \( n_x \times 1 \) vector of endogenous forward-looking variables that are physically free to jump at \( t \).

Let \( u_t \) be a vector of government instruments.

The \( z_t \) vector is inherited physically from the past.

But \( x_t \) is inherited as a consequence of decisions made by the Stackelberg planner at time \( t = 0 \).

Included in \( x_t \) might be prices and quantities that adjust instantaneously to clear markets at time \( t \).

Let \( y_t = \begin{bmatrix} z_t \\ x_t \end{bmatrix} \)

Define the governments one-period loss function\(^1\)

\[
    r(y, u) = y' R y + u' Q u
\]  

(6.1)

Subject to an initial condition for \( z_0 \), but not for \( x_0 \), a government wants to maximize

\[
    -\sum_{t=0}^{\infty} \beta^t r(y_t, u_t)
\]

(6.2)

The government makes policy in light of the model

\[
    \begin{bmatrix} I \\ G_{21} \\ G_{22} \end{bmatrix} \begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \tilde{A}_{11} & \tilde{A}_{12} \\ \tilde{A}_{21} & \tilde{A}_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + \tilde{B} u_t
\]

(6.3)

We assume that the matrix on the left is invertible, so that we can multiply both sides of (6.3) by its inverse to obtain

\[
    \begin{bmatrix} z_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} z_t \\ x_t \end{bmatrix} + B u_t
\]

(6.4)

or

\[
    y_{t+1} = Ay_t + Bu_t
\]

(6.5)

The private sectors behavior is summarized by the second block of equations of (6.4) or (6.5).

---

\(^1\) The problem assumes that there are no cross products between states and controls in the return function. A simple transformation converts a problem whose return function has cross products into an equivalent problem that has no cross products. For example, see [HS08] (chapter 4, pp. 72-73).
These equations typically include the first-order conditions of private agents optimization problem (i.e., their Euler equations).

These Euler equations summarize the forward-looking aspect of private agents behavior and express how their time $t$ decisions depend on government actions at times $s \geq t$.

When combined with a stability condition to be imposed below, the Euler equations summarize the private sectors best response to the sequence of actions by the government.

The government maximizes (6.2) by choosing sequences $\{u_t, x_t, z_{t+1}\}_{t=0}^{\infty}$ subject to (6.5) and an initial condition for $z_0$.

Note that we have an initial condition for $z_0$ but not for $x_0$.

$x_0$ is among the variables to be chosen at time 0 by the Stackelberg leader.

The government uses its understanding of the responses restricted by (6.5) to manipulate private sector actions.

To indicate the features of the Stackelberg leaders problem that make $x_t$ a vector of forward-looking variables, write the second block of system (6.3) as

$$x_t = \phi_1 z_t + \phi_2 z_{t+1} + \phi_3 u_t + \phi_0 x_{t+1}, \quad (6.6)$$

where $\phi_0 = A^{-1}_{22} G_{22}$.

The models we study in this chapter typically satisfy

*Forward-Looking Stability Condition* The eigenvalues of $\phi_0$ are bounded in modulus by $\beta^{-5}$. This stability condition makes equation (6.6) explosive if solved backwards but stable if solved forwards.

See the appendix of chapter 2 of [LS18].

So we solve equation (6.6) forward to get

$$x_t = \sum_{j=0}^{\infty} \phi_0^j \left[ \phi_1 z_{t+j} + \phi_2 z_{t+j+1} + \phi_3 u_{t+j} \right], \quad (6.7)$$

In choosing $u_t$ for $t \geq 1$ at time 0, the government takes into account how future $z$ and $u$ affect earlier $x$ through equation (6.7).

The lecture on history dependent policies analyzes an example about *Ramsey taxation* in which, as is typical in such problems, the last $n_x$ equations of (6.4) or (6.5) constitute *implementability constraints* that are formed by the Euler equations of a competitive fringe or private sector.

A *certainty equivalence principle* allows us to work with a nonstochastic model (see *LQ dynamic programming*).

That is, we would attain the same decision rule if we were to replace $x_{t+1}$ with the forecast $E_t x_{t+1}$ and to add a shock process $C \epsilon_{t+1}$ to the right side of (6.5), where $\epsilon_{t+1}$ is an IID random vector with mean zero and identity covariance matrix.
Let $s^t$ denote the history of any variable $s$ from $0$ to $t$.

[MS85], [HR85], [PL92], [Sar87], [Pea92], and others have all studied versions of the following problem:

**Problem S:** The Stackelberg problem is to maximize (6.2) by choosing an $x_0$ and a sequence of decision rules, the time $t$ component of which maps a time $t$ history of the natural state $z^t$ into a time $t$ decision $u_t$ of the Stackelberg leader.

The Stackelberg leader chooses this sequence of decision rules once and for all at time $t = 0$.

Another way to say this is that he commits to this sequence of decision rules at time 0.

The maximization is subject to a given initial condition for $z_0$.

But $x_0$ is among the objects to be chosen by the Stackelberg leader.

The optimal decision rule is history dependent, meaning that $u_t$ depends not only on $z_t$ but at $t \geq 1$ also on lags of $z$.

History dependence has two sources: (a) the government's ability to commit$^2$ to a sequence of rules at time 0 as in the lecture on history dependent policies, and (b) the forward-looking behavior of the private sector embedded in the second block of equations (6.4) as exhibited by (6.7).

### 6.1.3 Solving the Stackelberg Problem

**Some Basic Notation**

For any vector $a_t$, define $\tilde{a}_t = [a_t, a_{t+1}, \ldots]$.

Define a feasible set of $(\tilde{y}_0, \tilde{u}_0)$ sequences

$$
\Omega(y_0) = \{(\tilde{y}_1, \tilde{u}_0) : y_{t+1} = Ay_t + Bu_t, \forall t \geq 0\}
$$

Note that in the definition of $\Omega(y_0)$, $y_0$ is taken as given.

Eventually, the $x_0$ component of $y_0$ will be chosen, though it is taken as given in $\Omega(y_0)$.

**Two Subproblems**

Once again we use backward induction.

We express the Stackelberg problem in terms of two subproblems:

Subproblem 1 is solved by a continuation Stackelberg leader at each date $t \geq 1$.

Subproblem 2 is solved the Stackelberg leader at $t = 0$.

$^2$ The government would make different choices were it to choose sequentially, that is, were it to select its time $t$ action at time $t$. See the lecture on history dependent policies.
Subproblem 1

\[ v(y_0) = \max_{(\bar{y}, \bar{u}) \in \Omega(y_0)} \left( \sum_{t=0}^{\infty} \beta^t r(y_t, u_t) \right) \] (6.8)

Subproblem 2

\[ w(z_0) = \max_{x_0} v(y_0) \] (6.9)

Subproblem 1 takes the vector of forward-looking variables \( x_0 \) as given

Subproblem 2 optimizes over \( x_0 \)

The value function \( w(z_0) \) tells the value of the Stackelberg plan as a function of the vector of natural state variables at time 0, \( z_0 \)

Two Bellman equations

We now describe Bellman equations for \( v(y) \) and \( w(z_0) \)

Subproblem 1

The value function \( v(y) \) in subproblem 1 satisfies the Bellman equation

\[ v(y) = \max_{u, y^*} \left\{ -r(y, u) + \beta v(y^*) \right\} \] (6.10)

where the maximization is subject to

\[ y^* = Ay + Bu \] (6.11)

and \( y^* \) denotes next periods value.

Substituting \( v(y) = -y'Py \) into Bellman equation (6.10) gives

\[ -y'Py = \max_{u, y^*} \left\{ -y'Ry - u'Qu - \beta y'^*Py^* \right\} \]

which as in lecture linear regulator gives rise to the algebraic matrix Riccati equation

\[ P = R + \beta A'PA - \beta^2 A'PB(Q + \beta B'PB)^{-1}B'PA \] (6.12)
and the optimal decision rule coefficient vector

$$F = \beta (Q + \beta B'PB)^{-1} B'PA,$$  

(6.13)

where the optimal decision rule is

$$u_t = -F y_t.$$  

(6.14)

**Subproblem 2**

The value function \(v(y_0)\) satisfies

$$v(y_0) = -z'_0 P_{11} z_0 - 2x'_0 P_{21} z_0 - x'_0 P_{22} x_0$$  

(6.15)

where

$$P = \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix}$$

We find an optimal \(x_0\) by equating to zero the gradient of \(v(y_0)\) with respect to \(x_0\):

$$-2P_{21} z_0 - 2P_{22} x_0 = 0,$$

which implies that

$$x_0 = -P_{22}^{-1} P_{21} z_0.$$  

(6.16)

**Summary**

We solve the Stackelberg problem by

- formulating a particular optimal linear regulator
- solving the associated matrix Riccati equation (6.12) for \(P\)
- computing \(F\)
- then partitioning \(P\) to obtain representation (6.16)
Manifestation of time inconsistency

We have seen that for \( t \geq 0 \) the optimal decision rule for the Stackelberg leader has the form

\[ u_t = -F y_t \]

or

\[ u_t = f_{11} z_t + f_{12} x_t \]

where for \( t \geq 1 \), \( x_t \) is effectively a state variable, albeit not a natural one, inherited from the past.

The means that for \( t \geq 1 \), \( x_t \) is not a function of \( z_t \) only (though it is at \( t = 0 \)) and that \( x_t \) exerts an independent influence on \( u_t \).

The situation is different at \( t = 0 \).

For \( t = 0 \), the optimal choice of \( x_0 = -P_{22}^{-1} P_{21} z_0 \) described in equation (6.16) implies that

\[ u_0 = (f_{11} - f_{12} P_{22}^{-1} P_{21}) z_0 \quad (6.17) \]

So for \( t = 0 \), \( u_0 \) is a linear function of the natural state variable \( z_0 \) only.

But for \( t \geq 0 \), \( x_t \neq -P_{22}^{-1} P_{21} z_t \).

Nor does \( x_t \) equal any other linear combination of \( z_t \) for \( t \geq 1 \).

This means that \( x_t \) has an independent role in shaping \( u_t \) for \( t \geq 1 \).

All of this means that the Stackelberg leaders decision rule at \( t \geq 1 \) differs from its decision rule at \( t = 0 \).

As indicated at the beginning of this lecture, this difference is a symptom of the time inconsistency of the optimal Stackelberg plan.

6.1.4 Shadow prices

The history dependence of the governments plan can be expressed in the dynamics of Lagrange multipliers \( \mu_x \) on the last \( n_x \) equations of (6.3) or (6.4).

These multipliers measure the cost today of honoring past government promises about current and future settings of \( u \).

We shall soon show that as a result of optimally choosing \( x_0 \), it is appropriate to initialize the multipliers to zero at time \( t = 0 \).

This is true because at \( t = 0 \), there are no past promises about \( u \) to honor.

But the multipliers \( \mu_x \) take nonzero values thereafter, reflecting future costs to the government of confirming the private sectors earlier expectations about its time \( t \) actions.

From the linear regulator lecture, the formula \( \mu_t = P y_t \) for the vector of shadow prices on the transition equations is

\[ \mu_t = \begin{bmatrix} \mu_z \cr \mu_x \end{bmatrix} \]
The shadow price $\mu_{xt}$ on the forward-looking variables $x_t$ evidently equals

$$\mu_{xt} = P_{21}z_t + P_{22}x_t.$$  \hfill (6.18)

So (6.16) is equivalent with

$$\mu_{x0} = 0.$$  \hfill (6.19)

### 6.1.5 A Large Firm With a Competitive Fringe

As an example, this section studies the equilibrium of an industry with a large firm that acts as a Stackelberg leader with respect to a competitive fringe.

Sometimes the large firm is called the monopolist even though there are actually many firms in the industry.

The industry produces a single nonstorable homogeneous good, the quantity of which is chosen in the previous period.

One large firm produces $Q_t$ and a representative firm in a competitive fringe produces $q_t$.

The representative firm in the competitive fringe acts as a price taker and chooses sequentially.

The large firm commits to a policy at time 0, taking into account its ability to manipulate the price sequence, both directly through the effects of its quantity choices on prices, and indirectly through the responses of the competitive fringe to its forecasts of prices.

The costs of production are $C_t = eQ_t^2 + .5gQ_t^2 + .5c(Q_{t+1} - Q_t)^2$ for the large firm and $\sigma_t = dq_t + .5hq_t^2 + .5c(q_{t+1} - q_t)^2$ for the competitive firm, where $d > 0, e > 0, c > 0, g > 0, h > 0$ are cost parameters.

There is a linear inverse demand curve

$$p_t = A_0 - A_1(Q_t + \overline{q}_t) + v_t,$$  \hfill (6.20)

where $A_0, A_1$ are both positive and $v_t$ is a disturbance to demand governed by

$$v_{t+1} = \rho v_t + C_t \epsilon_{t+1}.$$  \hfill (6.21)

where $|\rho| < 1$ and $\epsilon_{t+1}$ is an IID sequence of random variables with mean zero and variance 1.

In (6.20), $\overline{q}_t$ is equilibrium output of the representative competitive firm.

In equilibrium, $\overline{q}_t = q_t$, but we must distinguish between $q_t$ and $\overline{q}_t$ in posing the optimum problem of a competitive firm.

---

[3][HS08] (chapter 16), uses this model as a laboratory to illustrate an equilibrium concept featuring robustness in which at least one of the agents has doubts about the stochastic specification of the demand shock process.
The competitive fringe

The representative competitive firm regards \( \{ p_t \}_{t=0}^{\infty} \) as an exogenous stochastic process and chooses an output plan to maximize

\[
E_0 \sum_{t=0}^{\infty} \beta^t \{ p_t q_t - \sigma_t \}, \quad \beta \in (0, 1)
\]

subject to \( q_0 \) given, where \( E_t \) is the mathematical expectation based on time \( t \) information

Let \( i_t = q_{t+1} - q_t \)

We regard \( i_t \) as the representative firms control at \( t \)

The first-order conditions for maximizing (6.22) are

\[
i_t = E_t \beta i_{t+1} - c^{-1} \beta h q_{t+1} + c^{-1} \beta E_t (p_{t+1} - d)
\]

for \( t \geq 0 \)

We appeal to a certainty equivalence principle to justify working with a non-stochastic version of (6.23) formed by dropping the expectation operator and the random term \( \epsilon_{t+1} \) from (6.21)

We use a method of [Sar79] and [Tow83]

We shift (6.20) forward one period, replace conditional expectations with realized values, use (6.20) to substitute for \( p_{t+1} \) in (6.23), and set \( q_t = \bar{q}_t \) and \( i_t = \bar{i}_t \) for all \( t \geq 0 \) to get

\[
\bar{i}_t = \beta \bar{i}_{t+1} - c^{-1} \beta h \bar{q}_{t+1} + c^{-1} \beta (A_0 - d) - c^{-1} \beta A_1 \bar{q}_{t+1} + c^{-1} \beta A_1 Q_{t+1} + c^{-1} \beta v_{t+1}
\]

Given sufficiently stable sequences \( \{ Q_t, v_t \} \), we could solve (6.24) and \( \bar{i}_t = \bar{q}_{t+1} - \bar{q}_t \) to express the competitive fringes output sequence as a function of the (tail of the) monopolists output sequence

(This would be a version of representation (6.7))

It is this feature that makes the monopolists problem fail to be recursive in the natural state variables \( \bar{q}, Q \)

The monopolist arrives at period \( t > 0 \) facing the constraint that it must confirm the expectations about its time \( t \) decision upon which the competitive fringe based its decisions at dates before \( t \)

The monopolists problem

The monopolist views the sequence of the competitive firms Euler equations as constraints on its own opportunities

They are implementability constraints on the monopolists choices

---

4 They used this method to compute a rational expectations competitive equilibrium. Their key step was to eliminate price and output by substituting from the inverse demand curve and the production function into the firms first-order conditions to get a difference equation in capital.
Including the implementability constraints, we can represent the constraints in terms of the transition law facing the monopolist:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
A_0 - d & 1 & -A_1 & -A_1 & h \end{bmatrix}
\begin{bmatrix}
v_{t+1} \\
Q_{t+1} \\
\bar{q}_{t+1} \\
\bar{u}_t \\
i_t + 1 \end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & \rho & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & c \end{bmatrix} \begin{bmatrix}
v_t \\
Q_t \\
\bar{q}_t \\
\bar{u}_t \\
i_t \end{bmatrix} + \begin{bmatrix}
1 \\
0 \\
1 \\
0 \\
0 \end{bmatrix} u_t \tag{6.25}
\]

where \( u_t = Q_{t+1} - Q_t \) is the control of the monopolist at time \( t \)

The last row portrays the implementability constraints (6.24)

Represent (6.25) as

\[
y_{t+1} = Ay_t + Bu_t \tag{6.26}
\]

Although we have included the competitive fringes choice variable \( \bar{t}_t \) as a component of the state \( y_t \) in the monopolists transition law (6.26), \( \bar{t}_t \) is actually a jump variable

Nevertheless, the analysis above implies that the solution of the large firms problem is encoded in the Riccati equation associated with (6.26) as the transition law

Lets decode it

To match our general setup, we partition \( y_t \) as \( y'_t = \begin{bmatrix} z'_t & x'_t \end{bmatrix} \) where \( z'_t = \begin{bmatrix} 1 & v_t & Q_t & \bar{q}_t \end{bmatrix} \) and \( x_t = \bar{t}_t \)

The monopolists problem is

\[
\max_{\{u_t, p_t, Q_t, \bar{q}_t, \bar{t}_t\}} \sum_{t=0}^{\infty} \beta^t \left\{ p_t Q_t - C_t \right\}
\]

subject to the given initial condition for \( z_0 \), equations (6.20) and (6.24) and \( \bar{t}_t = \bar{q}_{t+1} - \bar{q}_t \), as well as the laws of motion of the natural state variables \( z \)

Notice that the monopolist in effect chooses the price sequence, as well as the quantity sequence of the competitive fringe, albeit subject to the restrictions imposed by the behavior of consumers, as summarized by the demand curve (6.20) and the implementability constraint (6.24) that describes the best responses of firms in the competitive fringe

By substituting (6.20) into the above objective function, the monopolists problem can be expressed as

\[
\max_{\{u_t\}} \sum_{t=0}^{\infty} \beta^t \left\{ (A_0 - A_1 (\bar{q}_t + Q_t) + v_t)Q_t - eQ_t - 0.5gQ_t^2 - 0.5cu_t^2 \right\} \tag{6.27}
\]

subject to (6.26)

This can be written
\[
\max_{\{u_t\}} - \sum_{t=0}^{\infty} \beta^t \{ y'_t R y_t + u'_t Q u_t \} 
\]  
(6.28)

subject to (6.26) where

\[
R = - \begin{bmatrix}
0 & 0 & \frac{A_0 - \epsilon}{2} & 0 & 0 \\
0 & 0 & -\frac{A_0 - \epsilon}{2} & 0 & 0 \\
0 & 0 & -A_1 - .5g & -\frac{A_1}{2} & 0 \\
0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

and \(Q = \frac{\epsilon}{2}\)

Under the Stackelberg plan, \(u_t = -F y_t\), which implies that the evolution of \(y\) under the Stackelberg plan as

\[
\bar{y}_{t+1} = (A - BF) \bar{y}_t 
\]  
(6.29)

where \(\bar{y}_t = \begin{bmatrix} 1 & v_t & Q_t & \bar{q}_t & \bar{r}_t \end{bmatrix}'\)

**Recursive formulation of a followers problem**

We now make use of a Big \(K\), little \(k\) trick (see rational expectations equilibrium) to formulate a recursive version of a followers problem cast in terms of an ordinary Bellman equation

The individual firm faces \(\{p_t\}\) as a price taker and believes

\[
p_t = a_0 - a_1 Q_t - a_1 \bar{q}_t + v_t 
\equiv E_p [\bar{y}_t] 
\]  
(6.30)

(Please remember that \(\bar{q}_t\) is a component of \(\bar{y}_t\))

From the point of view of a representative firm in the competitive fringe, \(\{\bar{y}_t\}\) is an exogenous process

A representative fringe firm wants to forecast \(\bar{y}\) because it wants to forecast what it regards as the exogenous price process \(\{p_t\}\)

Therefore it wants to forecast the determinants of future prices

- future values of \(Q\) and
- future values of \(\bar{q}\)

An individual follower firm confronts state \(\begin{bmatrix} \bar{y}_t & q_t \end{bmatrix}'\) where \(q_t\) is its current output as opposed to \(\bar{q}\) within \(\bar{y}\)

It believes that it chooses future values of \(q_t\) but not future values of \(\bar{q}_t\)

(This is an application of a Big \(K\), little \(k\) idea)

The follower faces law of motion
\[
\begin{bmatrix}
\bar{y}_{t+1} \\
q_{t+1}
\end{bmatrix} =
\begin{bmatrix}
A - BF & 0 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
\bar{y}_t \\
q_t
\end{bmatrix} +
\begin{bmatrix}
0 \\
1
\end{bmatrix} t
\]

(6.31)

We calculated \( F \) and therefore \( A - BF \) earlier.

We can restate the optimization problem of the representative competitive firm.

The firm takes \( \bar{y}_t \) as an exogenous process and chooses an output plan \( \{q_t\} \) to maximize

\[
E_0 \sum_{t=0}^{\infty} \beta^t \{ p_t q_t - \sigma_t \}, \quad \beta \in (0, 1)
\]

subject to \( q_0 \) given the law of motion (6.29) and the price function (6.30) and where the costs are still \( \sigma_t = dq_t + .5hq_t^2 + .5c(q_{t+1} - q_t)^2 \)

The representative firms problem is a linear-quadratic dynamic programming problem with matrices \( A_s, B_s, Q_s, R_s \) that can be constructed easily from the above information.

The representative firms decision rule can be represented as

\[
i_t = -F_s
\begin{bmatrix}
1 \\
v_t \\
Q_t \\
\bar{y}_t \\
i_t \\
q_t
\end{bmatrix}
\]

(6.32)

Now let's stare at the decision rule (6.32) for \( i_t \), apply Big K, little k logic again, and ask what we want in order to verify a recursive representation of a representative followers choice problem.

- We want decision rule (6.32) to have the property that \( i_t = \bar{i}_t \) when we evaluate it at \( q_t = \bar{y}_t \)

We inherit these desires from a Big K, little k logic.

Here we apply a Big K, little k logic in two parts to make the representative firm be representative after solving the representative firms optimization problem.

- We want \( q_t = \bar{q}_t \)
- We want \( i_t = \bar{i}_t \)

**Numerical example**

We computed the optimal Stackelberg plan for parameter settings \( A_0, A_1, \rho, C_e, c, d, e, g, h, \beta = 100, 1, .8, 2, 1, 20, 20, .2, .2, .95 \)^5

^5 These calculations were performed by functions located in `dyn_stack/oligopoly.jl`.

6.1. Dynamic Stackelberg Problems 903
For these parameter values the monopolists decision rule is

\[ u_t = (Q_{t+1} - Q_t) = \begin{bmatrix} 83.98 & 0.78 & -0.95 & -1.31 & -2.07 \end{bmatrix} \begin{bmatrix} v_t \\ Q_t \\ \bar{q}_t \\ \bar{i}_t \end{bmatrix} \]

for \( t \geq 0 \)

and

\[ x_0 \equiv i_0 = \begin{bmatrix} 31.08 & 0.29 & -0.15 & -0.56 \end{bmatrix} \begin{bmatrix} v_0 \\ Q_0 \\ \bar{q}_0 \end{bmatrix} \]

For this example, starting from \( z_0 = \begin{bmatrix} 1 & v_0 & Q_0 & \bar{q}_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 25 & 46 \end{bmatrix} \), the monopolist chooses to set \( i_0 = 1.43 \)

That choice implies that

- \( i_1 = 0.25 \), and
- \( z_1 = \begin{bmatrix} 1 & v_1 & Q_1 & \bar{q}_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 21.83 & 47.43 \end{bmatrix} \)

A monopolist who started from the initial conditions \( \tilde{z}_0 = z_1 \) would set \( i_0 = 1.10 \) instead of .25 as called for under the original optimal plan.

The preceding little calculation reflects the time inconsistency of the monopolists optimal plan.

The recursive representation of the decision rule for a representative fringe firm is

\[ i_t = \begin{bmatrix} 0 & 0 & 0 & .34 & 1 & -34 \end{bmatrix} \begin{bmatrix} v_t \\ Q_t \\ \bar{q}_t \\ \bar{i}_t \\ q_t \end{bmatrix} \]

which we have computed by solving the appropriate linear-quadratic dynamic programming problem described above.

Notice that, as expected, \( i_t = \bar{i}_t \) when we evaluate this decision rule at \( q_t = \bar{q}_t \).

### 6.1.6 Concluding Remarks

This lecture is our first encounter with a class of problems in which optimal decision rules are history dependent.\(^6\)

We shall encounter other examples in lectures `optimal taxation with state-contingent debt` and `optimal taxation without state-contingent debt`.

Many more examples of such problems are described in chapters 20-24 of [LS18].

---

\(^6\) For another application of the techniques in this lecture and how they related to the method recommended by [KP80b], please see this lecture.
6.1.7 Exercises

Exercise 1

There is no uncertainty

For $t \geq 0$, a monetary authority sets the growth of (the log of) money according to

$$ m_{t+1} = m_t + u_t \quad (6.33) $$

subject to the initial condition $m_0 > 0$ given

The demand for money is

$$ m_t - p_t = -\alpha (p_{t+1} - p_t) \quad (6.34) $$

where $\alpha > 0$ and $p_t$ is the log of the price level

Equation (6.33) can be interpreted as the Euler equation of the holders of money

a. Briefly interpret how (6.33) makes the demand for real balances vary inversely with the expected rate of inflation. Temporarily (only for this part of the exercise) drop (6.33) and assume instead that $\{m_t\}$ is a given sequence satisfying $\sum_{t=0}^{\infty} m_t^2 < +\infty$. Solve the difference equation (6.33) forward to express $p_t$ as a function of current and future values of $m_s$. Note how future values of $m$ influence the current price level.

At time 0, a monetary authority chooses (commits to) a possibly history-dependent strategy for setting $\{u_t\}_{t=0}^{\infty}$

The monetary authority orders sequences $\{m_t, p_t\}_{t=0}^{\infty}$ according to

$$ -\sum_{t=0}^{\infty} .95^t \left[ (p_t - \bar{p})^2 + u_t^2 + .00001 m_t^2 \right] \quad (6.35) $$

Assume that $m_0 = 10, \alpha = 5, \bar{p} = 1$

b. Please briefly interpret this problem as one where the monetary authority wants to stabilize the price level, subject to costs of adjusting the money supply and some implementability constraints. (We include the term $.00001 m_t^2$ for purely technical reasons that you need not discuss.)

c. Please write and run a Python program to find the optimal sequence $\{u_t\}_{t=0}^{\infty}$

d. Display the optimal decision rule for $u_t$ as a function of $u_{t-1}, m_t, m_{t-1}$

e. Compute the optimal $\{m_t, p_t\}_t$ sequence for $t = 0, \ldots, 10$

Hints:

- The optimal $\{m_t\}$ sequence must satisfy $\sum_{t=0}^{\infty} (.95)^t m_t^2 < +\infty$

- Code can be found in the file lqcontrol.jl from the QuantEcon.jl package that implements the optimal linear regulator

6.1. Dynamic Stackelberg Problems 905
Exercise 2

A representative consumer has quadratic utility functional

\[ \sum_{t=0}^{\infty} \beta^t \{ -0.5(b - c_t)^2 \} \]  

(6.36)

where \( \beta \in (0, 1) \), \( b = 30 \), and \( c_t \) is time \( t \) consumption

The consumer faces a sequence of budget constraints

\[ c_t + a_{t+1} = (1 + r)a_t + y_t - \tau_t \]  

(6.37)

where

- \( a_t \) is the households holdings of an asset at the beginning of \( t \)
- \( r > 0 \) is a constant net interest rate satisfying \( \beta(1 + r) < 1 \)
- \( y_t \) is the consumers endowment at \( t \)

The consumers plan for \((c_t, a_{t+1})\) has to obey the boundary condition \( \sum_{t=0}^{\infty} \beta^t a_t^2 < +\infty \)

Assume that \( y_0, a_0 \) are given initial conditions and that \( y_t \) obeys

\[ y_t = \rho y_{t-1}, \quad t \geq 1, \]  

(6.38)

where \( |\rho| < 1 \). Assume that \( a_0 = 0, y_0 = 3 \), and \( \rho = .9 \)

At time 0, a planner commits to a plan for taxes \( \{\tau_t\}_{t=0}^{\infty} \)

The planner designs the plan to maximize

\[ \sum_{t=0}^{\infty} \beta^t \{ -0.5(c_t - b)^2 - \tau_t^2 \} \]  

(6.39)

over \( \{c_t, \tau_t\}_{t=0}^{\infty} \) subject to the implementability constraints in (6.37) for \( t \geq 0 \) and

\[ \lambda_t = \beta(1 + r)\lambda_{t+1} \]  

(6.40)

for \( t \geq 0 \), where \( \lambda_t \equiv (b - c_t) \)

a. Argue that (6.40) is the Euler equation for a consumer who maximizes (6.36) subject to (6.37), taking \( \{\tau_t\} \) as a given sequence

b. Formulate the planners problem as a Stackelberg problem

c. For \( \beta = .95, b = 30, \beta(1 + r) = .95 \), formulate an artificial optimal linear regulator problem and use it to solve the Stackelberg problem

d. Give a recursive representation of the Stackelberg plan for \( \tau_t \)
6.2 Optimal Taxation in an LQ Economy

Contents

- Optimal Taxation in an LQ Economy
  - Overview
  - The Ramsey Problem
  - Implementation
  - Examples
  - Exercises
  - Solutions

6.2.1 Overview

In this lecture we study optimal fiscal policy in a linear quadratic setting.

We slightly modify a well-known model of Robert Lucas and Nancy Stokey [LS83] so that convenient formulas for solving linear-quadratic models can be applied to simplify the calculations.

The economy consists of a representative household and a benevolent government.

The government finances an exogenous stream of government purchases with state-contingent loans and a linear tax on labor income.

A linear tax is sometimes called a flat-rate tax.

The household maximizes utility by choosing paths for consumption and labor, taking prices and the 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

6.2.2 The Ramsey Problem

We begin by outlining the key assumptions regarding technology, households and the government sector.

Technology

Labor can be converted one-for-one into a single, non-storable consumption good.

In the usual spirit of the LQ model, the amount of labor supplied in each period is unrestricted.

This is unrealistic, but helpful when it comes to solving the model.

Realistic labor supply can be induced by suitable parameter values.

Households

Consider a representative household who chooses a path \( \{\ell_t, c_t\} \) for labor and consumption to maximize

\[
\mathbb{E} \sum_{t=0}^{\infty} \beta^t \left[ (c_t - b_t)^2 + \ell_t^2 \right]
\]

subject to the budget constraint

\[
\mathbb{E} \sum_{t=0}^{\infty} \beta^t p^0_t \left[ d_t + (1 - \tau_t) \ell_t + s_t - c_t \right] = 0
\]

Here

- \( \beta \) is a discount factor in \((0, 1)\)
- \( p^0_t \) is a scaled Arrow-Debreu price at time 0 of history contingent goods at time \( t + j \)
- \( b_t \) is a stochastic preference parameter
- \( d_t \) is an endowment process
- \( \tau_t \) is a flat tax rate on labor income
- \( s_t \) is a promised time-\( t \) coupon payment on debt issued by the government
The scaled Arrow-Debreu price $p_0^t$ is related to the unscaled Arrow-Debreu price as follows. If we let $\pi_0^t(x^t)$ denote the probability (density) of a history $x^t = [x_t, x_{t-1}, \ldots, x_0]$ of the state $x^t$, then the Arrow-Debreu time $0$ price of a claim on one unit of consumption at date $t$, history $x^t$ would be

$$\frac{\beta^t p_0^t}{\pi_0^t(x^t)}$$

Thus, our scaled Arrow-Debreu price is the ordinary Arrow-Debreu price multiplied by the discount factor $\beta^t$ and divided by an appropriate probability.

The budget constraint (6.42) requires that the present value of consumption be restricted to equal the present value of endowments, labor income and coupon payments on bond holdings.

**Government**

The government imposes a linear tax on labor income, fully committing to a stochastic path of tax rates at time zero.

The government also issues state-contingent debt.

Given government tax and borrowing plans, we can construct a competitive equilibrium with distorting government taxes.

Among all such competitive equilibria, the Ramsey plan is the one that maximizes the welfare of the representative consumer.

**Exogenous Variables**

Endowments, government expenditure, the preference shock process $b_t$, and promised coupon payments on initial government debt $s_t$ are all exogenous, and given by

- $d_t = S_dx_t$
- $g_t = S_gx_t$
- $b_t = S_bx_t$
- $s_t = S_sx_t$

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
\[ c_t + g_t = d_t + \ell_t \]  
(6.43)

A labor-consumption process \( \{\ell_t, c_t\} \) is called feasible if (6.43) holds for all \( t \)

**Government budget constraint**

Where \( p_t^0 \) is again a scaled Arrow-Debreu price, the time zero government budget constraint is

\[
\mathbb{E} \sum_{t=0}^{\infty} \beta^t p_t^0 (s_t + g_t - \tau_t \ell_t) = 0
\]

(6.44)

**Equilibrium**

An equilibrium is a feasible allocation \( \{\ell_t, c_t\} \), a sequence of prices \( \{p_t^0\} \), and a tax system \( \{\tau_t\} \) such that

1. The allocation \( \{\ell_t, c_t\} \) is optimal for the household given \( \{p_t^0\} \) and \( \{\tau_t\} \)
2. The governments budget constraint (6.44) is satisfied

The Ramsey problem is to choose the equilibrium \( \{\ell_t, c_t, \tau_t, p_t^0\} \) that maximizes the households welfare

If \( \{\ell_t, c_t, \tau_t, p_t^0\} \) solves the Ramsey problem, then \( \{\tau_t\} \) is called the Ramsey plan

The solution procedure we adopt is

1. Use the first-order conditions from the household problem to pin down prices and allocations given \( \{\tau_t\} \)
2. Use these expressions to rewrite the government budget constraint (6.44) in terms of exogenous variables and allocations
3. Maximize the households objective function (6.41) subject to the constraint constructed in step 2 and the feasibility constraint (6.43)

The solution to this maximization problem pins down all quantities of interest

**Solution**

Step one is to obtain the first-conditions for the households problem, taking taxes and prices as given

Letting \( \mu \) be the Lagrange multiplier on (6.42), the first-order conditions are \( p_t^0 = (c_t - b_t)/\mu \) and \( \ell_t = (c_t - b_t)(1 - \tau_t) \)

Rearranging and normalizing at \( \mu = b_0 - c_0 \), we can write these conditions as

\[
p_t^0 = \frac{b_t - c_t}{b_0 - c_0} \quad \text{and} \quad \tau_t = 1 - \frac{\ell_t}{b_t - c_t}
\]

(6.45)
Substituting (6.45) into the governments budget constraint (6.44) yields

$$
\mathbb{E} \sum_{t=0}^{\infty} \beta^t \left[ (b_t - c_t)(s_t + g_t - \ell_t) + \ell_t^2 \right] = 0
$$

(6.46)

The Ramsey problem now amounts to maximizing (6.41) subject to (6.46) and (6.43)

The associated Lagrangian is

$$
\mathcal{L} = \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left\{ -\frac{1}{2} [ (c_t - b_t)^2 + \ell_t^2] + \lambda \left[ (b_t - c_t)(\ell_t - s_t - g_t) - \ell_t^2 \right] + \mu_t [d_t + \ell_t - c_t - g_t] \right\}
$$

(6.47)

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 (6.43) and working through the algebra, one can show that

$$
\ell_t = \tilde{\ell}_t - \nu m_t \quad \text{and} \quad c_t = \tilde{c}_t - \nu m_t
$$

(6.48)

where

- $\nu := \lambda / (1 + 2\lambda)$
- $\tilde{\ell}_t := (b_t - d_t + g_t) / 2$
- $\tilde{c}_t := (b_t + d_t - g_t) / 2$
- $m_t := (b_t - d_t - s_t) / 2$

Apart from $\nu$, all of these quantities are expressed in terms of exogenous variables

To solve for $\nu$, we can use the governments budget constraint again

The term inside the brackets in (6.46) is $(b_t - c_t)(s_t + g_t) - (b_t - c_t)\ell_t + \ell_t^2$

Using (6.48), the definitions above and the fact that $\tilde{\ell} = b - \tilde{c}$, this term can be rewritten as

$$(b_t - \tilde{c}_t)(g_t + s_t) + 2m_t^2(\nu^2 - \nu)$$

Reinserting into (6.46), we get

$$
\mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t (b_t - \tilde{c}_t)(g_t + s_t) \right\} + (\nu^2 - \nu) \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t 2m_t^2 \right\} = 0
$$

(6.49)

Although it might not be clear yet, we are nearly there because:

6.2. Optimal Taxation in an LQ Economy
• The two expectations terms in (6.49) 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 (6.48)
• Once we have the allocations, prices and the tax system can be derived from (6.45)

**Computing the Quadratic Term**

Let's consider how to obtain the term \( \nu \) in (6.49)

If we can compute the two expected geometric sums

\[
b_0 := \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t (b_t - \bar{c}_t) (g_t + s_t) \right\} \quad \text{and} \quad a_0 := \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t 2m_t^2 \right\}
\]

(6.50)

then the problem reduces to solving

\[b_0 + a_0 (\nu^2 - \nu) = 0\]

for \( \nu \)

Provided that \( 4b_0 < a_0 \), there is a unique solution \( \nu \in (0, 1/2) \), and a unique corresponding \( \lambda > 0 \)

Let's work out how to compute mathematical expectations in (6.50)

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 (6.50), the random variable \(2m_t^2\) can be written as

\[
\frac{1}{2} x_t' (S_b - S_d - S_s)' (S_b - S_d - S_s) x_t
\]

It follows that both objects of interest are special cases of the expression

\[
q(x_0) = \mathbb{E} \sum_{t=0}^{\infty} \beta^t x_t' H x_t
\]

(6.51)

where \( H \) is a matrix conformable to \( x_t \) and \( x_t' \) is the transpose of column vector \( x_t \)

Suppose first that \( \{x_t\} \) is the Gaussian VAR described above

In this case, the formula for computing \( q(x_0) \) is known to be \( q(x_0) = x_0'Qx_0 + v \), where

- \( Q \) is the solution to \( Q = H + \beta A'QA \), and
- \( v = \text{trace} \ (C'QC)\beta/(1 - \beta) \)

The first equation is known as a discrete Lyapunov equation, and can be solved using this function
Finite state Markov case

Next suppose that \( \{x_t\} \) is the discrete Markov process described above

Suppose further that each \( x_t \) takes values in the state space \( \{x_1, \ldots, x_N\} \subset \mathbb{R}^k \)

Let \( h: \mathbb{R}^k \to \mathbb{R} \) be a given function, and suppose that we wish to evaluate

\[
q(x_0) = E \sum_{t=0}^{\infty} \beta^t h(x_t) \quad \text{given} \quad x_0 = x^j
\]

For example, in the discussion above, \( h(x_t) = x'_t H x_t \)

It is legitimate to pass the expectation through the sum, leading to

\[
q(x_0) = \sum_{t=0}^{\infty} \beta^t (P^t h)[j]
\]  

Here

- \( P^t \) is the \( t \)-th power of the transition matrix \( P \)
- \( h \) is, with some abuse of notation, the vector \( (h(x^1), \ldots, h(x^N)) \)
- \( (P^t h)[j] \) indicates the \( j \)-th element of \( P^t h \)

It can be shown that (6.52) is in fact equal to the \( j \)-th element of the vector \( (I - \beta P)^{-1} h \)

This last fact is applied in the calculations below

Other Variables

We are interested in tracking several other variables besides the ones described above.

To prepare the way for this, we define

\[
p_{t+j} = \frac{b_{t+j} - c_{t+j}}{b_t - c_t}
\]

as the scaled Arrow-Debreu time \( t+j \) price of a history contingent claim on one unit of consumption at time \( t+j \)

These are prices that would prevail at time \( t \) if market were reopened at time \( t \)

These prices are constituents of the present value of government obligations outstanding at time \( t \), which can be expressed as

\[
B_t := \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j p_{t+j} (t_{t+j} \ell_{t+j} - g_{t+j})
\]  

(6.53)
Using our expression for prices and the Ramsey plan, we can also write $B_t$ as

$$B_t = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j (b_{t+j} - c_{t+j})(\ell_{t+j} - g_{t+j}) - \ell_{t+j}^2 / b_t - c_t$$

This version is more convenient for computation.

Using the equation

$$p_{t+j} = p_{t+1} \beta p_{t+j}$$

it is possible to verify that (6.53) implies that

$$B_t = (\tau_t \ell_t - g_t) + \mathbb{E}_t \beta \sum_{j=1}^{\infty} p_{t+j} (\tau_{t+j} \ell_{t+j} - g_{t+j})$$

and

$$B_t = (\tau_t \ell_t - g_t) + \beta \mathbb{E}_t p_{t+1}^t B_{t+1}$$

Define

$$R_t^{-1} := \mathbb{E}_t \beta p_{t+1}^t$$

$R_t$ is the gross 1-period risk-free rate for loans between $t$ and $t + 1$

**A Martingale**

We now want to study the following two objects, namely,

$$\pi_{t+1} := B_{t+1} - R_t [B_t - (\tau_t \ell_t - g_t)]$$

and the cumulation of $\pi_t$

$$\Pi_t := \sum_{s=0}^{t} \pi_t$$

The term $\pi_{t+1}$ is the difference between two quantities:

- $B_{t+1}$, the value of government debt at the start of period $t + 1$
- $R_t [B_t + g_t - \tau_t]$, which is what the government would have owed at the beginning of period $t + 1$ if it had simply borrowed at the one-period risk-free rate rather than selling state-contingent securities
Thus, \( \pi_{t+1} \) is the excess payout on the actual portfolio of state contingent government debt relative to an alternative portfolio sufficient to finance \( B_t + g_t - \tau_t \ell_t \) and consisting entirely of risk-free one-period bonds.

Use expressions (6.54) and (6.55) to obtain

\[
\pi_{t+1} = B_{t+1} - \frac{1}{\beta E_t p_{t+1}^t} \left[ \beta E_t p_{t+1}^t B_{t+1} \right]
\]

or

\[
\pi_{t+1} = B_{t+1} - \tilde{E}_t B_{t+1}
\]

(6.56)

where \( \tilde{E}_t \) is the conditional mathematical expectation taken with respect to a one-step transition density that has been formed by multiplying the original transition density with the likelihood ratio

\[
m_{t+1}^t = \frac{p_{t+1}^t}{E_t p_{t+1}^t}
\]

It follows from equation (6.56) that

\[
\tilde{E}_t \pi_{t+1} = \tilde{E}_t B_{t+1} - \tilde{E}_t B_{t+1} = 0
\]

which asserts that \( \{ \pi_{t+1} \} \) is a martingale difference sequence under the distorted probability measure, and that \( \{ \Pi_t \} \) is a martingale under the distorted probability measure.

In the tax-smoothing model of Robert Barro \([Bar79]\), government debt is a random walk.

In the current model, government debt \( \{ B_t \} \) is not a random walk, but the excess payoff \( \{ \Pi_t \} \) on it is

6.2.3 Implementation

The following code provides functions for

1. Solving for the Ramsey plan given a specification of the economy
2. Simulating the dynamics of the major variables

Description and clarifications are given below

```julia
#=
@author: Spencer Lyon <spencer.lyon@nyu.edu>

=#
using QuantEcon
using PyPlot
using LaTeXStrings

abstract type AbstractStochProcess end

struct ContStochProcess(TF <: AbstractFloat) <: AbstractStochProcess

6.2. Optimal Taxation in an LQ Economy
915
```
A::Matrix{TF}
C::Matrix{TF}

end

struct DiscreteStochProcess{TF <: AbstractFloat} <: AbstractStochProcess
  P::Matrix{TF}
  x_vals::Matrix{TF}
end

struct Economy{TF <: AbstractFloat, SP <: AbstractStochProcess} :: TF
  β::TF
  Sg::Matrix{TF}
  Sd::Matrix{TF}
  Sb::Matrix{TF}
  Ss::Matrix{TF}
  is_discrete::Bool
  proc::SP
end

struct Path{TF <: AbstractFloat}
  g::Vector{TF}
  d::Vector{TF}
  b::Vector{TF}
  s::Vector{TF}
  c::Vector{TF}
  l::Vector{TF}
  p::Vector{TF}
  τ::Vector{TF}
  rvn::Vector{TF}
  B::Vector{TF}
  R::Vector{TF}
  π::Vector{TF}
  ξ::Vector{TF}
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 ν = λ / (1 + 2*λ). Here ν is the
  solution to a quadratic equation a(ν^2 - ν) + b = 0 where
  a and b are expected discounted sums of quadratic forms of the state. =#
  Sm = Sb - Sd - Ss

  return g, d, b, s, Sm
end
function compute_allocation(econ::Economy, Sm::Array, ν::AbstractFloat, x::Array, b::Array)
    Sg, Sd, Sb, Ss = econ.Sg, econ.Sd, econ.Sb, econ.Ss
    # Solve for the allocation given ν and x
    Sc = 0.5 .* (Sb + Sd - Sg - ν .* Sm)
    Sl = 0.5 .* (Sb - Sd + Sg - ν .* Sm)
    c = squeeze(Sc * x, 1)
    l = squeeze(Sl * x, 1)
    p = squeeze((Sb - Sc) * x, 1)  # Price without normalization
    τ = 1 .- l ./ (b .- c)
    rvn = l .* τ
    return Sc, Sl, c, l, p, τ, rvn
end

function compute_ν(a0::AbstractFloat, b0::AbstractFloat)
    disc = a0^2 - 4a0 * b0
    if disc >= 0
        ν = 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 ν * (0.5 - ν) < 0
        println("Negative multiplier on the government budget constraint."")
        error("Government spending (economy.g) too low")
    end
    return ν
end

function compute_Π(B::Vector, R::Vector, rvn::Vector, g::Vector, ξ::Vector)
    Π = cumsum(π .* ξ)
    return π, Π
end

function compute_paths{TF <: AbstractFloat}(econ::Economy{TF, DiscreteStochProcess{TF}},)
    # simplify notation
    β, Sg, Sd, Sb, Ss = econ.β, econ.Sg, econ.Sd, econ.Sb, econ.Ss
    P, x_vals = econ.proc.P, econ.proc.x_vals

6.2. Optimal Taxation in an LQ Economy
mc = MarkovChain(P)
state = simulate(mc, T, init=1)
x = x_vals[:, state]

# Compute exogenous sequence

# Compute a0, b0

ns = size(P, 1)
F = eye(ns) - β .* 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_ν(a0, b0)

# Solve for the allocation given ν and x

Sc, Sl, c, l, p, τ, rvn = compute_allocation(econ, Sm, ν, 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 ./ (β .* H)
temp = squeeze(P[state, :] * ((Sb - Sc) * x_vals)', 2)
ξ = p[2:end] ./ temp[1:end-1]

# Compute π
π, Π = compute_Π(B, R, rvn, g, ξ)

Path(g, d, b, s, c, l, p, τ, rvn, B, R, π, Π, ξ)
end

function compute_paths{TF<:AbstractFloat}(econ::Economy{TF, ContStochProcess !{TF}}, T::Integer)

# Simplify notation

β, Sg, Sd, Sb, Ss = econ.β, econ.Sg, econ.Sd, econ.Sb, econ.Ss

# Generate an initial condition x0 satisfying x0 = A x0

nx, nx = size(A)
x0 = nullspace((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 = Matrix{TF}(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, β, x0)
H = (Sb - Sd + Sg) * (Sg + Ss)
b0 = 0.5 * var_quadratic_sum(A, C, H, β, x0)

# compute lagrange multiplier
ν = compute_ν(a0, b0)

# Solve for the allocation given ν and x
Sc, Sl, c, l, p, τ, rvn = compute_allocation(econ, Sm, ν, x, b)

# compute remaining variables
H = Sl' * Sl - (Sb - Sc) * (S1 - Sg)
L = Vector{TF}(T)
for t=1:T
    L[t] = var_quadratic_sum(A, C, H, β, x[:, t])
end
B = L . / p
Rinv = squeeze(β .* (Sb - Sc) * A * x, 1) . / p
R = 1 . / Rinv
AF1 = (Sb - Sc) * x[:, 2:end]
AF2 = (Sb - Sc) * A * x[:, 1:end-1]
ξ = AF1 . / AF2
ξ = squeeze(ξ, 1)

# compute π
π, Π = compute_Π(B, R, rvn, g, ξ)
Path(g, d, b, s, c, l, p, τ, rvn, B, R, π, Π, ξ)
end

function gen_fig_1(path::Path)
    T = length(path.c)
    figure(figsize=(12, 8))
    ax1 = subplot(2, 2, 1)
    ax1[:plot](path.rvn)
    ax1[:plot](path.g)
    ax1[:plot](path.c)
    ax1[:set_xlabel]("Time")
    ax1[:legend](["L"\$\tau_t \ell_t\$", L"Sg_t\$", L"Sc_t\$"])
The function `var_quadratic_sum` from QuantEcon.jl is for computing the value of (6.51) 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.

### 6.2.4 Examples

Let's look at two examples of usage.
The Continuous Case

Our first example adopts the VAR specification described above.

Regarding the primitives, we set

- \( \beta = 1/1.05 \)
- \( b_t = 2.135 \) and \( s_t = d_t = 0 \) for all \( t \)

Government spending evolves according to

\[
g_{t+1} - \mu_g = \rho(g_t - \mu_g) + C_g w_{g,t+1}
\]

with \( \rho = 0.7, \mu_g = 0.35 \) and \( C_g = \mu_g \sqrt{1 - \rho^2}/10 \)

Here's the code

```julia
# == Parameters == #
\beta = 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; 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(\beta, Sg, Sd, Sb, Ss, discrete, proc)
T = 50
path = compute_paths(econ, T)
gen_fig_1(path)
```

6.2. Optimal Taxation in an LQ Economy
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.
The Discrete Case

Our second example adopts a discrete Markov specification for the exogenous process

```python
# Parameters
beta = 1 / 1.05
P = [0.8 0.2 0.0
     0.0 0.5 0.5
     0.0 0.0 1.0]
```

See the original manuscript for comments and interpretation

6.2. Optimal Taxation in an LQ Economy
# Possible states of the world
# Each column is a state of the world. The rows are [g d b s l]

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(β, Sg, Sd, Sb, Ss, discrete, proc)
T = 15
path = compute_paths(econ, T)

gen_fig_1(path)

The call gen_fig_2(path) generates
gen_fig_2(path)

See the original manuscript for comments and interpretation

6.2. Optimal Taxation in an LQ Economy
6.2.5 Exercises

Exercise 1

Modify the VAR example *given above*, setting

\[ g_{t+1} - \mu_g = \rho(g_{t-3} - \mu_g) + C_g w_{g,t+1} \]

with \( \rho = 0.95 \) and \( C_g = 0.7 \sqrt{1 - \rho^2} \)

Produce the corresponding figures

6.2.6 Solutions

```julia
# Parameters
\beta = 1 / 1.05
\rho, mg = .95, .35
A = [0. 0. 0. \rho  mg*(1-\rho) ;
    1. 0. 0. 0. ;
    0. 1. 0. 0. ;
    0. 0. 1. 0. ;
    0. 0. 0. 0. ]
C[1, 1] = sqrt(1 - \rho^2) + mg / 8
Sg = [1. 0. 0. 0. 0.]
Sd = [0. 0. 0. 0. 0.]
Sb = [0. 0. 0. 0. 2.135]
Ss = [0. 0. 0. 0. 0.]
discrete = false
proc = ContStochProcess(A, C)
econ = Economy(\beta, Sg, Sd, Sb, Ss, discrete, proc)

T = 50
path = compute_paths(econ, T)

gen_fig_1(path)
```
6.2. Optimal Taxation in an LQ Economy
6.3 Optimal Taxation with State-Contingent Debt

Contents

- Optimal Taxation with State-Contingent Debt
  - Overview
  - A Competitive Equilibrium with Distorting Taxes
  - Recursive Formulation of the Ramsey problem
  - Examples
6.3.1 Overview

This lecture describes a celebrated model of optimal fiscal policy by Robert E. Lucas, Jr., and Nancy Stokey [LS83].

The model revisits classic issues about how to pay for a war.

Here a war means a more or less temporary surge in an exogenous government expenditure process.

The model features

- a government that must finance an exogenous stream of government expenditures with either
  - a flat rate tax on labor, or
  - purchases and sales from a full array of Arrow state-contingent securities
- a representative household that values consumption and leisure
- a linear production function mapping labor into a single good
- a Ramsey planner who at time $t = 0$ chooses a plan for taxes and trades of Arrow securities for all $t \geq 0$

After first presenting the model in a space of sequences, we shall represent it recursively in terms of two Bellman equations formulated along lines that we encountered in Dynamic Stackelberg models.

As in Dynamic Stackelberg models, to apply dynamic programming we shall define the state vector artfully.

In particular, we shall include forward-looking variables that summarize optimal responses of private agents to a Ramsey plan.

See Optimal taxation for an analysis within a linear-quadratic setting.

6.3.2 A Competitive Equilibrium with Distorting Taxes

For $t \geq 0$, a history $s^t = [s_t, s_{t-1}, \ldots, s_0]$ of an exogenous state $s_t$ has joint probability density $\pi_t(s^t)$

We begin by assuming that government purchases $g_t(s^t)$ at time $t \geq 0$ depend on $s^t$.

Let $c_t(s^t)$, $\ell_t(s^t)$, and $n_t(s^t)$ denote consumption, leisure, and labor supply, respectively, at history $s^t$ and date $t$.

A representative household is endowed with one unit of time that can be divided between leisure $\ell_t$ and labor $n_t$:

$$n_t(s^t) + \ell_t(s^t) = 1 \quad (6.57)$$

Output equals $n_t(s^t)$ and can be divided between $c_t(s^t)$ and $g_t(s^t)$:

$$c_t(s^t) + g_t(s^t) = n_t(s^t) \quad (6.58)$$

A representative households preferences over $\{c_t(s^t), \ell_t(s^t)\}_{t=0}^{\infty}$ are ordered by
\[
\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)]
\]  

(6.59)

where the utility function \( u \) is increasing, strictly concave, and three times continuously differentiable in both arguments.

The technology pins down a pre-tax wage rate to unity for all \( t, s^t \).

The government imposes a flat-rate tax \( \tau_t(s^t) \) on labor income at time \( t \), history \( s^t \).

There are complete markets in one-period Arrow securities.

One unit of an Arrow security issued at time \( t \) at history \( s^t \) and promising to pay one unit of time \( t+1 \) consumption in state \( s_{t+1} \) costs \( p_{t+1}(s_{t+1}|s^t) \).

The government issues one-period Arrow securities each period.

The government has a sequence of budget constraints whose time \( t \geq 0 \) component is

\[
g_t(s^t) = \tau_t(s^t)n_t(s^t) + \sum_{s_{t+1}} p_{t+1}(s_{t+1}|s^t) b_{t+1}(s_{t+1}|s^t) - b_t(s_t|s^{t-1})
\]

(6.60)

where

- \( p_{t+1}(s_{t+1}|s^t) \) is a competitive equilibrium price of one unit of consumption at date \( t+1 \) in state \( s_{t+1} \) at date \( t \) and history \( s^t \).
- \( b_t(s_t|s^{t-1}) \) is government debt falling due at time \( t \), history \( s^t \).

Government debt \( b_0(s_0) \) is an exogenous initial condition.

The representative household has a sequence of budget constraints whose time \( t \geq 0 \) component is

\[
c_t(s^t) + \sum_{s_{t+1}} p_t(s_{t+1}|s^t) b_{t+1}(s_{t+1}|s^t) = \left[ 1 - \tau_t(s^t) \right] n_t(s^t) + b_t(s_t|s^{t-1}) \quad \forall t \geq 0.
\]

(6.61)

A **government policy** is an exogenous sequence \( \{g(s_t)\}_{t=0}^{\infty} \), a tax rate sequence \( \{\tau_t(s^t)\}_{t=0}^{\infty} \), and a government debt sequence \( \{b_{t+1}(s^{t+1})\}_{t=0}^{\infty} \)

A **feasible allocation** is a consumption-labor supply plan \( \{c_t(s^t), n_t(s^t)\}_{t=0}^{\infty} \) that satisfies (6.58) at all \( t, s^t \).

A **price system** is a sequence of Arrow security prices \( \{p_{t+1}(s_{t+1}|s^t)\}_{t=0}^{\infty} \).

The household faces the price system as a price-taker and takes the government policy as given.

The household chooses \( \{c_t(s^t), \ell_t(s^t)\}_{t=0}^{\infty} \) to maximize (6.59) subject to (6.61) and (6.57) for all \( t, s^t \).

A **competitive equilibrium with distorting taxes** is a feasible allocation, a price system, and a government policy such that

- Given the price system and the government policy, the allocation solves the households optimization problem.
• Given the allocation, government policy, and price system, the government's budget constraint is satisfied for all $t, s^t$

Note: There are many competitive equilibria with distorting taxes

They are indexed by different government policies

The **Ramsey problem** or **optimal taxation problem** is to choose a competitive equilibrium with distorting taxes that maximizes (6.59)

### Arrow-Debreu Version of Price System

We find it convenient sometimes to work with the Arrow-Debreu price system that is implied by a sequence of Arrow securities prices

Let $q^0_t(s^t)$ be the price at time 0, measured in time 0 consumption goods, of one unit of consumption at time $t$, history $s^t$

The following recursion relates Arrow-Debreu prices $\{q^0_t(s^t)\}_{t=0}^{\infty}$ to Arrow securities prices

$$q^0_{t+1}(s^{t+1}) = p_{t+1}(s_{t+1}|s^t)q^0_t(s^t) \quad s.t. \quad q^0_0(s^0) = 1 \tag{6.62}$$

Arrow-Debreu prices are useful when we want to compress a sequence of budget constraints into a single intertemporal budget constraint, as we shall find it convenient to do below

### Primal Approach

We apply a popular approach to solving a Ramsey problem, called the **primal approach**

The idea is to use first-order conditions for household optimization to eliminate taxes and prices in favor of quantities, then pose an optimization problem cast entirely in terms of quantities

After Ramsey quantities have been found, taxes and prices can then be unwound from the allocation

The primal approach uses four steps:

1. Obtain first-order conditions of the households problem and solve them for $\{q^0_t(s^t), \tau_t(s^t)\}_{t=0}^{\infty}$ as functions of the allocation $\{c_t(s^t), n_t(s^t)\}_{t=0}^{\infty}$

2. Substitute these expressions for taxes and prices in terms of the allocation into the households present-value budget constraint

   • This intertemporal constraint involves only the allocation and is regarded as an implementability constraint

3. Find the allocation that maximizes the utility of the representative household (6.59) subject to the feasibility constraints (6.57) and (6.58) and the implementability condition derived in step 2

   • This optimal allocation is called the **Ramsey allocation**

4. Use the Ramsey allocation together with the formulas from step 1 to find taxes and prices
The Implementability Constraint

By sequential substitution of one one-period budget constraint (6.61) into another, we can obtain the household's present-value budget constraint:

\[
\sum_{t=0}^{\infty} \sum_{s^t} q^0_t(s^t) c_t(s^t) = \sum_{t=0}^{\infty} \sum_{s^t} q^0_t(s^t) [1 - \tau_t(s^t)] n_t(s^t) + b_0 \tag{6.63}
\]

\(\{q^0_t(s^t)\}_{t=1}^{\infty}\) can be interpreted as a time 0 Arrow-Debreu price system.

To approach the Ramsey problem, we study the households optimization problem.

First-order conditions for the households problem for \(\ell_t(s^t)\) and \(b_t(s_{t+1}|s^t)\), respectively, imply

\[
(1 - \tau_t(s^t)) = \frac{u_t(s^t)}{u_c(s^t)} \tag{6.64}
\]

and

\[
p_{t+1}(s_{t+1}|s^t) = \beta \pi(s_{t+1}|s^t) \left( \frac{u_c(s^{t+1})}{u_c(s^t)} \right) \tag{6.65}
\]

where \(\pi(s_{t+1}|s^t)\) is the probability distribution of \(s_{t+1}\) conditional on history \(s^t\).

Equation (6.65) implies that the Arrow-Debreu price system satisfies

\[
q^0_t(s^t) = \beta^t \pi_t(s^t) \frac{u_c(s^t)}{u_c(s^0)} \tag{6.66}
\]

Using the first-order conditions (6.64) and (6.65) to eliminate taxes and prices from (6.63), we derive the implementability condition

\[
\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) [u_c(s^t) c_t(s^t) - u_c(s^t) n_t(s^t)] - u_c(s^0) b_0 = 0. \tag{6.67}
\]

The Ramsey problem is to choose a feasible allocation that maximizes

\[
\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), 1 - n_t(s^t)] \tag{6.68}
\]

subject to (6.67)
Solution Details

First define a pseudo utility function

\[ V [c_t(s^t), n_t(s^t), \Phi] = u[c_t(s^t), 1 - n_t(s^t)] + \Phi \left[ u_c(s^t)c_t(s^t) - u_\ell(s^t)n_t(s^t) \right] \]  
(6.69)

where \( \Phi \) is a Lagrange multiplier on the implementability condition (6.63)

Next form the Lagrangian

\[ J = \sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) \left\{ V[c_t(s^t), n_t(s^t), \Phi] + \theta_t(s^t) \left[ n_t(s^t) - c_t(s^t) - g_t(s_t) \right] \right\} - \Phi u_\ell(0)b_0 \]  
(6.70)

where \( \{\theta_t(s^t); \forall s^t\}_{t \geq 0} \) is a sequence of Lagrange multipliers on the feasible conditions (6.58)

Given an initial government debt \( b_0 \), we want to maximize \( J \) with respect to \( \{c_t(s^t), n_t(s^t); \forall s^t\}_{t \geq 0} \) and to minimize with respect to \( \{\theta(s^t); \forall s^t\}_{t \geq 0} \)

The first-order conditions for the Ramsey problem for periods \( t \geq 1 \) and \( t = 0 \), respectively, are

\[
\begin{align*}
c_t(s^t): & \quad (1 + \Phi)u_c(s^t) + \Phi \left[ u_{cc}(s^t)c_t(s^t) - u_{\ell\ell}(s^t)n_t(s^t) \right] - \theta_t(s^t) = 0, \quad t \geq 1 \\
n_t(s^t): & \quad - (1 + \Phi)u_\ell(s^t) - \Phi \left[ u_{cc}(s^t)c_t(s^t) - u_{\ell\ell}(s^t)n_t(s^t) \right] + \theta_t(s^t) = 0, \quad t \geq 1
\end{align*}
\]  
(6.71)

and

\[
\begin{align*}
c_0(s^0, b_0): & \quad (1 + \Phi)u_c(s^0, b_0) + \Phi \left[ u_{cc}(s^0, b_0)c_0(s^0, b_0) - u_{\ell\ell}(s^0, b_0)n_0(s^0, b_0) \right] - \theta_0(s^0, b_0) \\
& - \Phi u_{cc}(s^0, b_0)b_0 = 0 \\
n_0(s^0, b_0): & \quad - (1 + \Phi)u_\ell(s^0, b_0) - \Phi \left[ u_{cc}(s^0, b_0)c_0(s^0, b_0) - u_{\ell\ell}(s^0, b_0)n_0(s^0, b_0) \right] + \theta_0(s^0, b_0) \\
& + \Phi u_{\ell\ell}(s^0, b_0)b_0 = 0
\end{align*}
\]  
(6.72)

Please note how these first-order conditions differ between \( t = 0 \) and \( t \geq 1 \)

It is instructive to use first-order conditions (6.71) for \( t \geq 1 \) to eliminate the multipliers \( \theta_t(s^t) \)

For convenience, we suppress the time subscript and the index \( s^t \) and obtain

\[
(1 + \Phi)u_c(c, 1 - c - g) + \Phi [cu_{cc}(c, 1 - c - g) - (c + g)u_{\ell\ell}(c, 1 - c - g)] \\
= (1 + \Phi)u_\ell(c, 1 - c - g) + \Phi [cu_{cc}(c, 1 - c - g) - (c + g)u_{\ell\ell}(c, 1 - c - g)]
\]  
(6.73)

where we have imposed conditions (6.57) and (6.58)

Equation (6.73) is one equation that can be solved to express the unknown \( c \) as a function of the exogenous variable \( g \)

We also know that time \( t = 0 \) quantities \( c_0 \) and \( n_0 \) satisfy

6.3. Optimal Taxation with State-Contingent Debt
(1 + \Phi)u_c(c, 1 - c - g) + \Phi\left[ cu_{cc}(c, 1 - c - g) - (c + g)u_{c\ell}(c, 1 - c - g) \right]
= (1 + \Phi)u_c(c, 1 - c - g) + \Phi\left[ cu_{c\ell}(c, 1 - c - g) - (c + g)u_{\ell\ell}(c, 1 - c - g) \right] + \Phi(u_{cc} - u_{c\ell})b_0
\tag{6.74}

Notice that a counterpart to \( b_0 \) does not appear in (6.73), so \( c \) does not depend on it for \( t \geq 1 \)

But things are different for time \( t = 0 \)

An analogous argument for the \( t = 0 \) equations (6.72) leads to one equation that can be solved for \( c_0 \) as a function of the pair \((g(s_0), b_0)\)

These outcomes mean that the following statement would be true even when government purchases are history-dependent functions \( g_t(s^t) \) of the history of \( s^t \)

**Proposition:** If government purchases are equal after two histories \( s^t \) and \( \tilde{s}^\tau \) for \( t, \tau \geq 0 \), i.e., if

\[
g_t(s^t) = g^\tau(\tilde{s}^\tau) = g
\]

then it follows from (6.73) that the Ramsey choices of consumption and leisure, \((c_t(s^t), \ell_t(s^t))\) and \((c_j(\tilde{s}^\tau), \ell_j(\tilde{s}^\tau))\), are identical

The proposition asserts that the optimal allocation is a function of the currently realized quantity of government purchases \( g \) only and does not depend on the specific history that preceded that realization of \( g \)

**The Ramsey Allocation for a Given \( \Phi \)**

Temporarily take \( \Phi \) as given

We shall compute \( c_0(s^0, b_0) \) and \( n_0(s^0, b_0) \) from the first-order conditions (6.72)

Evidently, for \( t \geq 1 \), \( c \) and \( n \) depend on the time \( t \) realization of \( g \) only

But for \( t = 0 \), \( c \) and \( n \) depend on both \( g_0 \) and the governments initial debt \( b_0 \)

Thus, while \( b_0 \) influences \( c_0 \) and \( n_0 \), there appears no analogous variable \( b_t \) that influences \( c_t \) and \( n_t \) for \( t \geq 1 \)

The absence of \( b_t \) as a determinant of the Ramsey allocation for \( t \geq 1 \) and its presence for \( t = 0 \) is a symptom of the time-inconsistency of a Ramsey plan

\( \Phi \) has to take a value that assures that the household and the governments budget constraints are both satisfied at a candidate Ramsey allocation and price system associated with that \( \Phi \)

**Further Specialization**

At this point, it is useful to specialize the model in the following ways

We assume that \( s \) is governed by a finite state Markov chain with states \( s \in [1, \ldots, S] \) and transition matrix \( \Pi \), where

\[
\Pi(s'|s) = \text{Prob}(s_{t+1} = s'|s_t = s)
\]
Also, assume that government purchases \( g \) are an exact time-invariant function \( g(s) \) of \( s \).

We maintain these assumptions throughout the remainder of this lecture.

**Determining \( \Phi \)**

We complete the Ramsey plan by computing the Lagrange multiplier \( \Phi \) on the implementability constraint (6.67).

Government budget balance restricts \( \Phi \) via the following line of reasoning:

The households first-order conditions imply

\[
(1 - \tau_t(s^t)) = \frac{u_t(s^t)}{u_c(s^t)} \tag{6.75}
\]

and the implied one-period Arrow securities prices

\[
p_{t+1}(s_{t+1}|s^t) = \beta \Pi(s_{t+1}|s_t) \frac{u_c(s^{t+1})}{u_c(s^t)} \tag{6.76}
\]

Substituting from (6.75), (6.76), and the feasibility condition (6.58) into the recursive version (6.61) of the household budget constraint gives

\[
u_c(s^t)[n_t(s^t) - g_t(s^t)] + \beta \sum_{s_{t+1}} \Pi(s_{t+1}|s_t) u_c(s^{t+1}) b_{t+1}(s_{t+1}|s^t) = u_t(s^t)n_t(s^t) + u_c(s^t)b_t(s_t|s^{t-1}) \tag{6.77}
\]

Define \( x_t(s^t) = u_c(s^t)b_t(s_t|s^{t-1}) \)

Notice that \( x_t(s^t) \) appears on the right side of (6.77) while \( \beta \) times the conditional expectation of \( x_{t+1}(s^{t+1}) \) appears on the left side.

Hence the equation shares much of the structure of a simple asset pricing equation with \( x_t \) being analogous to the price of the asset at time \( t \).

We learned earlier that for a Ramsey allocation \( c_t(s^t), n_t(s^t) \) and \( b_t(s_t|s^{t-1}) \), and therefore also \( x_t(s^t) \), are each functions of \( s_t \) only, being independent of the history \( s^{t-1} \) for \( t \geq 1 \).

That means that we can express equation (6.77) as

\[
u_c(s)[n(s) - g(s)] + \beta \sum_{s'} \Pi(s'|s)x'(s') = u_t(s)n(s) + x(s) \tag{6.78}
\]

where \( s' \) denotes a next period value of \( s \) and \( x'(s') \) denotes a next period value of \( x \).

Equation (6.78) is easy to solve for \( x(s) \) for \( s = 1, \ldots, S \).
If we let $\vec{n}, \vec{g}, \vec{x}$ denote $S \times 1$ vectors whose $i$th elements are the respective $n, g,$ and $x$ values when $s = i$, and let $\Pi$ be the transition matrix for the Markov state $s$, then we can express (6.78) as the matrix equation

$$\vec{u}_c(\vec{n} - \vec{g}) + \beta \Pi \vec{x} = \vec{u}_l \vec{n} + \vec{x}$$  
(6.79)

This is a system of $S$ linear equations in the $S \times 1$ vector $x$, whose solution is

$$\vec{x} = (I - \beta \Pi)^{-1}[\vec{u}_c(\vec{n} - \vec{g}) - \vec{u}_l \vec{n}]$$  
(6.80)

In these equations, by $\vec{u}_c \vec{n}$, for example, we mean element-by-element multiplication of the two vectors.

After solving for $\vec{x}$, we can find $b(s_t | s^{t-1})$ in Markov state $s_t = s$ from $b(s) = \frac{x(s)}{u_c(s)}$ or the matrix equation

$$\vec{b} = \frac{\vec{x}}{\vec{u}_c}$$  
(6.81)

where division here means element-by-element division of the respective components of the $S \times 1$ vectors $\vec{x}$ and $\vec{u}_c$.

Here is a computational algorithm:

1. Start with a guess for the value for $\Phi$, then use the first-order conditions and the feasibility conditions to compute $c(s_t), n(s_t)$ for $s \in [1, \ldots, S]$ and $c_0(s_0, b_0)$ and $n_0(s_0, b_0)$, given $\Phi$
   - these are $2(S + 1)$ equations in $2(S + 1)$ unknowns
2. Solve the $S$ equations (6.80) for the $S$ elements of $\vec{x}$
   - these depend on $\Phi$
3. Find a $\Phi$ that satisfies

$$u_{c,0}b_0 = u_{c,0}(n_0 - g_0) - u_{l,0}n_0 + \beta \sum_{s=1}^{S} \Pi(s | s_0)x(s)$$  
(6.82)

by gradually raising $\Phi$ if the left side of (6.82) exceeds the right side and lowering $\Phi$ if the left side is less than the right side
4. After computing a Ramsey allocation, recover the flat tax rate on labor from (6.64) and the implied one-period Arrow securities prices from (6.65)

In summary, when $g_t$ is a time invariant function of a Markov state $s_t$, a Ramsey plan can be constructed by solving $3S + 3$ equations in $S$ components each of $\vec{c}, \vec{n},$ and $\vec{x}$ together with $n_0, c_0$, and $\Phi$
Time Inconsistency

Let \( \{ \tau_t(s^t) \}_{t=0}^{\infty}, \{ b_{t+1}(s_{t+1}|s^t) \}_{t=0}^{\infty} \) be a time 0, state \( s_0 \) Ramsey plan.

Then \( \{ \tau_j(s^j) \}_{j=t}^{\infty}, \{ b_{j+1}(s_{j+1}|s^j) \}_{j=t}^{\infty} \) is a time \( t \), history \( s^t \) continuation of a time 0, state \( s_0 \) Ramsey plan.

A time \( t \), history \( s^t \) Ramsey plan is a Ramsey plan that starts from initial conditions \( s^t, b_t(s_t|s^{t-1}) \).

A time \( t \), history \( s^t \) continuation of a time 0, state 0 Ramsey plan is not a time \( t \), history \( s^t \) Ramsey plan.

The means that a Ramsey plan is not time consistent.

Another way to say the same thing is that a Ramsey plan is time inconsistent.

The reason is that a continuation Ramsey plan takes \( u_{ct} b_t(s_t|s^{t-1}) \) as given, not \( b_t(s_t|s^{t-1}) \).

We shall discuss this more below.

Specification with CRRA Utility

In our calculations below and in a subsequent lecture based on an extension of the Lucas-Stokey model by Aiyagari, Marcet, Sargent, and Seppälä (2002) [AMSS02], we shall modify the one-period utility function assumed above.

(We adopted the preceding utility specification because it was the one used in the original [LS83] paper.)

We will modify their specification by instead assuming that the representative agent has utility function

\[
u(c, n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma}\]

where \( \sigma > 0 \), \( \gamma > 0 \).

We continue to assume that

\[c_t + g_t = n_t\]

We eliminate leisure from the model.

We also eliminate Lucas and Stokey's restriction that \( \ell_t + n_t \leq 1 \).

We replace these two things with the assumption that labor \( n_t \in [0, +\infty] \).

With these adjustments, the analysis of Lucas and Stokey prevails once we make the following replacements

\[
u_{\ell}(c, \ell) \sim -u_{\ell}(c, n)
\]
\[
u_c(c, \ell) \sim u_{c}(c, n)
\]
\[
u_{\ell,\ell}(c, \ell) \sim u_{\ell\ell}(c, n)
\]
\[
u_{c,\ell}(c, \ell) \sim u_{c,\ell}(c, n)
\]
\[u_{c,c}(c, \ell) \sim 0\]

With these understandings, equations (6.73) and (6.74) simplify in the case of the CRRA utility function.

They become

6.3. Optimal Taxation with State-Contingent Debt
\[
(1 + \Phi)[u_c(c) + u_n(c + g)] + \Phi[c u_{cc}(c) + (c + g) u_{nn}(c + g)] = 0 \tag{6.83}
\]

and

\[
(1 + \Phi)[u_c(c_0) + u_n(c_0 + g_0)] + \Phi[c_0 u_{cc}(c_0) + (c_0 + g_0) u_{nn}(c_0 + g_0)] - \Phi u_{cc}(c_0)b_0 = 0 \tag{6.84}
\]

In equation (6.83), it is understood that \( c \) and \( g \) are each functions of the Markov state \( s \).

In addition, the time \( t = 0 \) budget constraint is satisfied at \( c_0 \) and initial government debt \( b_0 \):

\[
b_0 + g_0 = \tau_0(c_0 + g_0) + \frac{\bar{b}}{R_0} \tag{6.85}
\]

where \( R_0 \) is the gross interest rate for the Markov state \( s_0 \) that is assumed to prevail at time \( t = 0 \) and \( \tau_0 \) is the time \( t = 0 \) tax rate.

In equation (6.85), it is understood that

\[
\tau_0 = 1 - \frac{u_{l,0}}{u_{c,0}}
\]

\[
R_0 = \beta \sum_{s=1}^{S} \Pi(s|s_0) \frac{u_c(s)}{u_{c,0}}
\]

**Sequence Implementation**

The above steps are implemented in a type called `SequentialAllocation`

```julia
using QuantEcon
using NLsolve
using NLopt

import QuantEcon.simulate

mutable struct Model{TF <: AbstractFloat,
TM <: AbstractMatrix{TF},
TV <: AbstractVector{TF}}

β::TF
Π::TM
G::TV
Θ::TV
transfers::Bool
U::Function
Uc::Function
Ucc::Function
Un::Function
```

938 Chapter 6. Dynamic Programming Squared
6.3. Optimal Taxation with State-Contingent Debt

```julia
Unn::Function
    n_less_than_one::Bool
end

struct SequentialAllocation{TP <: Model,
    TI <: Integer,
    TV <: AbstractVector}
    model::TP
    mc::MarkovChain
    S::TI
    cFB::TV
    nFB::TV
    ΞFB::TV
    zFB::TV
end

function SequentialAllocation(model::Model)
    β, Π, G, Θ = model.β, model.Π, model.G, model.Θ
    mc = MarkovChain(Π)
    S = size(Π, 1)  # Number of states
    # Now find the first best allocation
    cFB, nFB, ΞFB, zFB = find_first_best(model, S, 1)
    return SequentialAllocation(model, mc, S, cFB, nFB, ΞFB, zFB)
end

function find_first_best(model::Model, S::Integer, version::Integer)
    if version !== 1 || version !== 2
        throw(ArgumentError("version must be 1 or 2"))
    end
    β, Θ, Uc, Un, G, Π = 
    model.β, model.Θ, model.Uc, model.Un, model.G, model.Π
    function res!(out, z)
        c = z[1:S]
        n = z[S+1:end]
        out[1:S] = Θ .* Uc(c, n) + Un(c, n)
        out[S+1:end] = Θ .* n - c - G
    end
    res = nlsolve(res!, 0.5 * ones(2 * S))
    if converged(res) == false
        error("Could not find first best")
    end
end
```
if version == 1
    cFB = res.zero[1:S]
    nFB = res.zero[S+1:end]
    ξFB = Uc(cFB, nFB)  # Multiplier on the resource constraint
    zFB = vcat(cFB, nFB, ξFB)
    return cFB, nFB, ξFB, zFB
elseif version == 2
    cFB = res.zero[1:S]
    nFB = res.zero[S+1:end]
    IFB = Uc(cFB, nFB) .* cFB + Un(cFB, nFB) .* nFB
    xFB = \(\text{eye}(S) - \beta \cdot \Pi, \text{IFB})
    zFB = [vcat(cFB[s], xFB[s], xFB) for s in 1:S]
    return cFB, nFB, IFB, xFB, zFB
end
end

function time1_allocation(pas::SequentialAllocation, μ::Real)
    model, S = pas.model, pas.S
    Θ, β, Π, G, Uc, Ucc, Un, Unn =
        model.Θ, model.β, model.Π, model.G,
        model.Uc, model.Ucc, model.Un, model.Unn
    function FOC!(out, z::Vector)
        c = z[1:S]
        n = z[S+1:2S]
        ξ = z[2S+1:end]
        out[1:S] = Uc(c, n) - μ * (Ucc(c, n) .* c + Uc(c, n)) - ξ
        out[S+1:2S] = Un(c, n) - μ * (Unn(c, n) .* n + Un(c, n)) + Θ .* ξ
        out[2S+1:end] = Θ .* n - c - G
        return out
    end
    # Find the root of the FOC
    res = nlsolve(FOC!, pas.zFB)
    if res.f_converged == false
        error("Could not find LS allocation.")
    end
    z = res.zero
    c, n, ξ = z[1:S], z[S+1:2S], z[2S+1:end]
    # Now compute x
    I = Uc(c, n) .* c + Un(c, n) .* n
    x = \(\text{eye}(S) - \beta \cdot \text{model.Π, I})
    return c, n, x, ξ
end

# Finds the optimal allocation given initial government debt `B_0` and state `s_0`

function time0_allocation(pas::SequentialAllocation, B_::AbstractFloat, s_0::Integer)
    model = pas.model
    Π, Θ, G, β = model.Π, model.Θ, model.G, model.β
    Uc, Ucc, Un, Unn = model.Uc, model.Ucc, model.Un, model.Unn

    # First order conditions of planner's problem
    function FOC!(out, z)
        μ, c, n, Ξ = z[1], z[2], z[3], z[4]
        xprime = time1_allocation(pas, μ)[3]
        out .= vcat(
            Uc(c, n) .* (c - B_) + Un(c, n) .* n + β * dot(Π[s_0, :], xprime),
            Uc(c, n) - μ * (Ucc(c, n) .* (c - B_) + Uc(c, n)) - Ξ,
            Un(c, n) - μ * (Unn(c, n) .* n + Un(c, n)) + Θ[s_0] .* Ξ,
            (Θ .* n - c - G)[s_0]
        )
    end

    # Find root
    res = nlsolve(FOC!, [0.0, pas.cFB[s_0], pas.nFB[s_0], pas.ΞFB[s_0]])
    if res.f_converged == false
        error("Could not find time 0 LS allocation."")
    end
    return (res.zero...)
end

function time1_value(pas::SequentialAllocation, B_::Real)
    model = pas.model
    c, n, x, Ξ = time1_allocation(pas, μ)[3]
    U_val = model.U.(c, n)
    V = \(\text{eye}(\text{pas.S}) - \text{model.β} \times \text{model.Π}, \text{U_val})
    return c, n, x, V
end

function T(model::Model, c::Union{Real, Vector}, n::Union{Real, Vector})
    Uc, Un = model.Uc.(c, n), model.Un.(c, n)
    return 1+Un.\(model.Θ \times Uc)
end

function simulate(pas::SequentialAllocation, B_::AbstractFloat, s_0::Integer, T::Integer,
sHist::Union(Vector, Void)=nothing

model = pas.model
Π, β, Uc = model.Π, model.β, model.Uc

if sHist == nothing
    sHist = QuantEcon.simulate(pas.mc, T, init=s_0)
end

chHist = zeros(T)
nHist = zeros(T)
BHist = zeros(T)
THist = zeros(T)
µHist = zeros(T)
RHist = zeros(T-1)

# time 0
µ, cHist[1], nHist[1], _ = time0_allocation(pas, B_, s_0)
THist[1] = T(pas.model, cHist[1], nHist[1])[s_0]
BHist[1] = B_
µHist[1] = µ

# time 1 onward
for t in 2:T
    c, n, x, Ξ = time1_allocation(pas, µ)
u_c = Uc(c,n)
s = sHist[t]
THist[t] = T(pas.model, c, n)[s]
Eu_c = dot(Π[sHist[t-1],:], u_c)
cHist[t], nHist[t], BHist[t] = c[s], n[s], x[s] / u_c[s]
RHist[t-1] = Uc(cHist[t-1], nHist[t-1]) / (β + Eu_c)
µHist[t] = µ
end

return chHist, nHist, BHist, THist, sHist, µHist, RHist
end

""
Bellman equation for the continuation of the Lucas-Stokey Problem
""

mutable struct BellmanEquation{TP <: Model,
    TI <: Integer,
    TV <: AbstractVector,
    TM <: AbstractMatrix{TV},
    TVV <: AbstractVector{TV}}

model::TP
S::TI
xbar::TV
time_0::Bool
z0::TM
cFB::TV
nFB::TV
xBFB::TV
zFB::TVV
end

"""
```julia
function BellmanEquation(model::Model, xgrid::AbstractVector, policies0::Vector)
    S = size(model, 1)  # Number of states
    xbar = [minimum(xgrid), maximum(xgrid)]
    time_0 = false
    cf, nf, xprimef = policies0
    z0 = [vcat(cf[s](x), nf[s](x), [xprimef[s, sprime](x) for sprime in 1:S]) for x in xgrid, s in 1:S]
    cFB, nFB, IFB, xFB, zFB = find_first_best(model, S, 2)
    return BellmanEquation(model, S, xbar, time_0, z0, cFB, nFB, xFB, zFB)
end

function get_policies_time1(T::BellmanEquation, i_x::Integer, x::AbstractFloat, s::Integer, Vf::AbstractArray)
    model, S = T.model, T.S
    β, θ, G, Π = model.β, model.θ, model.G, model.Π
    U, Uc, Un = model.U, model.Uc, model.Un

    function objf(z::Vector, grad)
        c, xprime = z[1], z[2:end]
        n = c + G[s]
        Vprime = [Vf[sprime](xprime[sprime]) for sprime in 1:S]
        return -(U(c, n) + β * dot(Π[s, :], Vprime))
    end

    function cons(z::Vector, grad)
        c, xprime = z[1], z[2:end]
        n = c + G[s]
        return x - Uc(c, n) * c - Un(c, n) * n - β * dot(Π[s, :], xprime)
    end

    lb = vcat(0, T.xbar[1] * ones(S))
    ub = vcat(1 - G[s], T.xbar[2] * ones(S))
    opt = Opt(:LN_COBYLA, length(T.z0[i_x, s]) - 1)
    min_objective!(opt, objf)
    equality_constraint!(opt, cons)
    lower_bounds!(opt, lb)
    upper_bounds!(opt, ub)
    maxeval!(opt, 300)
    maxtime!(opt, 10)
    init = vcat(T.z0[i_x, s][1], T.z0[i_x, s][3:end])
    for (i, val) in enumerate(init)
        if val > ub[i]
            init[i] = ub[i]
        elseif val < lb[i]
            init[i] = lb[i]
        end
    end
    (minf, minx, ret) = optimize(opt, init)
end
```

6.3. Optimal Taxation with State-Contingent Debt
6.3.3 Recursive Formulation of the Ramsey problem

\( x_t(s^t) = u_c(s^t) b_t(s_t | s^{t-1}) \) in equation (6.77) appears to be a purely forward-looking variable.

But \( x_t(s^t) \) is also a natural candidate for a state variable in a recursive formulation of the Ramsey problem.
**Intertemporal Delegation**

To express a Ramsey plan recursively, we imagine that a time 0 Ramsey planner is followed by a sequence of continuation Ramsey planners at times \( t = 1, 2, \ldots \).

A continuation Ramsey planner has a different objective function and faces different constraints than a Ramsey planner.

A key step in representing a Ramsey plan recursively is to regard the marginal utility scaled government debts \( x_t(s^t) = u_c(s^t)b_t(s^t|s^{t-1}) \) as predetermined quantities that continuation Ramsey planners at times \( t \geq 1 \) are obligated to attain.

Continuation Ramsey planners do this by choosing continuation policies that induce the representative household to make choices that imply that \( u_c(s^t)b_t(s^t|s^{t-1}) = x_t(s^t) \).

A time \( t \geq 1 \) continuation Ramsey planner delivers \( x_t \) by choosing a suitable \( n_t, c_t \) pair and a list of \( s_{t+1} \)-contingent continuation quantities \( x_{t+1} \) to bequeath to a time \( t + 1 \) continuation Ramsey planner.

A time \( t \geq 1 \) continuation Ramsey planner faces \( x_t, s_t \) as state variables.

But the time 0 Ramsey planner faces \( b_0 \), not \( x_0 \), as a state variable.

Furthermore, the Ramsey planner cares about \( (c_0(s_0), \ell_0(s_0)) \), while continuation Ramsey planners do not.

The time 0 Ramsey planner hands \( x_1 \) as a function of \( s_1 \) to a time 1 continuation Ramsey planner.

These lines of delegated authorities and responsibilities across time express the continuation Ramsey planners obligations to implement their parts of the original Ramsey plan, designed once-and-for-all at time 0.

**Two Bellman Equations**

After \( s_t \) has been realized at time \( t \geq 1 \), the state variables confronting the time \( t \) continuation Ramsey planner are \( (x_t, s_t) \).

- Let \( V(x, s) \) be the value of a **continuation Ramsey plan** at \( x_t = x, s_t = s \) for \( t \geq 1 \).
- Let \( W(b, s) \) be the value of a **Ramsey plan** at time 0 at \( b_0 = b \) and \( s_0 = s \).

We work backwards by presenting a Bellman equation for \( V(x, s) \) first, then a Bellman equation for \( W(b, s) \).

**The Continuation Ramsey Problem**

The Bellman equation for a time \( t \geq 1 \) continuation Ramsey planner is

\[
V(x, s) = \max_{n(x'(s'))} u(n - g(s), 1 - n) + \beta \sum_{s' \in S} \Pi(s'|s) V(x', s')
\]  
(6.86)

where maximization over \( n \) and the \( S \) elements of \( x'(s') \) is subject to the single implementability constraint for \( t \geq 1 \).
\[ x = u_c(n - g(s)) - u_l n + \beta \sum_{s' \in S} \Pi(s'|s)x'(s') \]  

(6.87)

Here \( u_c \) and \( u_l \) are today's values of the marginal utilities.

For each given value of \( x, s \), the continuation Ramsey planner chooses \( n \) and an \( x'(s') \) for each \( s' \in S \).

Associated with a value function \( V(x, s) \) that solves Bellman equation (6.86) are \( S + 1 \) time-invariant policy functions:

\[ n_t = f(x_t, s_t), \quad t \geq 1 \]
\[ x_{t+1}(s_{t+1}) = h(s_{t+1}; x_t, s_t), \quad s_{t+1} \in S, \quad t \geq 1 \]  

(6.88)

The Ramsey Problem

The Bellman equation for the time 0 Ramsey planner is

\[ W(b_0, s_0) = \max_{n_0, x'(s_1)} u(n_0 - g_0, 1 - n_0) + \beta \sum_{s_1 \in S} \Pi(s_1|s_0)V(x'(s_1), s_1) \]  

(6.89)

where maximization over \( n_0 \) and the \( S \) elements of \( x'(s_1) \) is subject to the time 0 implementability constraint:

\[ u_{c,0}b_0 = u_{c,0}(n_0 - g_0) - u_{l,0}n_0 + \beta \sum_{s_1 \in S} \Pi(s_1|s_0)x'(s_1) \]  

(6.90)

coming from restriction (6.82).

Associated with a value function \( W(b_0, n_0) \) that solves Bellman equation (6.89) are \( S + 1 \) time 0 policy functions:

\[ n_0 = f_0(b_0, s_0) \]
\[ x_1(s_1) = h_0(s_1; b_0, s_0) \]  

(6.91)

Notice the appearance of state variables \((b_0, s_0)\) in the time 0 policy functions for the Ramsey planner as compared to \((x_t, s_t)\) in the policy functions (6.88) for the time \( t \geq 1 \) continuation Ramsey planners.

The value function \( V(x_t, s_t) \) of the time \( t \) continuation Ramsey planner equals \( E_t \sum_{\tau = t}^{\infty} \beta^{\tau-t}u(c_t, l_t) \), where the consumption and leisure processes are evaluated along the original time 0 Ramsey plan.

First-Order Conditions

Attach a Lagrange multiplier \( \Phi_1(x, s) \) to constraint (6.87) and a Lagrange multiplier \( \Phi_0 \) to constraint (6.82).

Time \( t \geq 1 \): the first-order conditions for the time \( t \geq 1 \) constrained maximization problem on the right side of the continuation Ramsey planners Bellman equation (6.86) are
\[ \beta \Pi(s'|s)V_x(x', s') - \beta \Pi(s'|s)\Phi_1 = 0 \]  \hspace{1cm} (6.92)

for \( x'(s') \) and

\[ (1 + \Phi_1)(u_c - u_l) + \Phi_1 [n(u_ll - u_lc) + (n - g(s))(u_{cc} - u_{lc})] = 0 \]  \hspace{1cm} (6.93)

for \( n \)

Given \( \Phi_1 \), equation (6.93) is one equation to be solved for \( n \) as a function of \( s \) (or of \( g(s) \))

Equation (6.92) implies \( V_x(x', s') = \Phi_1 \), while an envelope condition is \( V_x(x, s) = \Phi_1 \), so it follows that

\[ V_x(x', s') = V_x(x, s) = \Phi_1(x, s) \]  \hspace{1cm} (6.94)

Time \( t = 0 \): For the time 0 problem on the right side of the Ramsey planners Bellman equation (6.89), first-order conditions are

\[ V_x(x(s_1), s_1) = \Phi_0 \]  \hspace{1cm} (6.95)

for \( x(s_1), s_1 \in \mathcal{S} \), and

\[ (1 + \Phi_0)(u_{c,0} - u_{n,0}) + \Phi_0 [n_0(u_{ll,0} - u_{lc,0}) + (n_0 - g(s_0))(u_{cc,0} - u_{lc,0})] - \Phi_0 (u_{cc,0} - u_{lc,0})b_0 = 0 \]  \hspace{1cm} (6.96)

Notice similarities and differences between the first-order conditions for \( t \geq 1 \) and for \( t = 0 \)

An additional term is present in (6.96) except in three special cases

- \( b_0 = 0 \), or
- \( u_c \) is constant (i.e., preferences are quasi-linear in consumption), or
- initial government assets are sufficiently large to finance all government purchases with interest earnings from those assets, so that \( \Phi_0 = 0 \)

Except in these special cases, the allocation and the labor tax rate as functions of \( s_t \) differ between dates \( t = 0 \) and subsequent dates \( t \geq 1 \)

Naturally, the first-order conditions in this recursive formulation of the Ramsey problem agree with the first-order conditions derived when we first formulated the Ramsey plan in the space of sequences
State Variable Degeneracy

Equations (6.95) and (6.96) imply that $\Phi_0 = \Phi_1$ and that

$$V_x(x_t, s_t) = \Phi_0$$

(6.97)

for all $t \geq 1$

When $V$ is concave in $x$, this implies state-variable degeneracy along a Ramsey plan in the sense that for $t \geq 1$, $x_t$ will be a time-invariant function of $s_t$

Given $\Phi_0$, this function mapping $s_t$ into $x_t$ can be expressed as a vector $\vec{x}$ that solves equation (6.90) for $n$ and $c$ as functions of $g$ that are associated with $\Phi = \Phi_0$

Manifestations of Time Inconsistency

While the marginal utility adjusted level of government debt $x_t$ is a key state variable for the continuation Ramsey planners at $t \geq 1$, it is not a state variable at time 0

The time 0 Ramsey planner faces $b_0$, not $x_0 = u_{c,0}b_0$, as a state variable

The discrepancy in state variables faced by the time 0 Ramsey planner and the time $t \geq 1$ continuation Ramsey planners captures the differing obligations and incentives faced by the time 0 Ramsey planner and the time $t \geq 1$ continuation Ramsey planners

• The time 0 Ramsey planner is obligated to honor government debt $b_0$ measured in time 0 consumption goods

• The time 0 Ramsey planner can manipulate the value of government debt as measured by $u_{c,0}b_0$

• In contrast, time $t \geq 1$ continuation Ramsey planners are obligated not to alter values of debt, as measured by $u_{c,t}b_t$, that they inherit from a preceding Ramsey planner or continuation Ramsey planner

When government expenditures $g_t$ are a time invariant function of a Markov state $s_t$, a Ramsey plan and associated Ramsey allocation feature marginal utilities of consumption $u_c(s_t)$ that, given $\Phi$, for $t \geq 1$ depend only on $s_t$, but that for $t = 0$ depend on $b_0$ as well

This means that $u_c(s_t)$ will be a time invariant function of $s_t$ for $t \geq 1$, but except when $b_0 = 0$, a different function for $t = 0$

This in turn means that prices of one period Arrow securities $p_{t+1}(s_{t+1}|s_t) = p(s_{t+1}|s_t)$ will be the same time invariant functions of $(s_{t+1}, s_t)$ for $t \geq 1$, but a different function $p_0(s_1|s_0)$ for $t = 0$, except when $b_0 = 0$

The differences between these time 0 and time $t \geq 1$ objects reflect the Ramsey planners incentive to manipulate Arrow security prices and, through them, the value of initial government debt $b_0$

Recursive Implementation

The above steps are implemented in a type called RecursiveAllocation
Compute the planner's allocation by solving Bellman equation.

```julia
struct RecursiveAllocation{TP <: Model, TI <: Integer,
    TVg <: AbstractVector, TVv <: AbstractVector,
    TVp <: AbstractArray}  
    model::TP 
    mc::MarkovChain 
    S::TI 
    T::BellmanEquation 
    μgrid::TVg 
    xgrid::TVg 
    Vf::TVv 
    policies::TVp 
end

function RecursiveAllocation(model::Model, μgrid::AbstractArray)  
    mc = MarkovChain(model.Π) 
    G = model.G 
    S = size(model.Π, 1) # Number of states 
    # Now find the first best allocation 
    Vf, policies, T, xgrid = solve_time1_bellman(model, μgrid) 
    T.time_0 = true # Bellman equation now solves time 0 problem 
    return RecursiveAllocation(model, mc, S, T, μgrid, xgrid, Vf, policies)
end

function solve_time1_bellman{TF <: AbstractFloat}(model::Model{TF}, μgrid::AbstractArray)  
    μgrid0 = μgrid 
    # First get initial fit 
    PP = SequentialAllocation(model) 
    c = Matrix{TF}(length(μgrid), 2) 
    n = Matrix{TF}(length(μgrid), 2) 
    x = Matrix{TF}(length(μgrid), 2) 
    V = Matrix{TF}(length(μgrid), 2) 
    for (i, μ) in enumerate(μgrid0)  
        c[i, :], n[i, :], x[i, :], V[i, :] = timel_value(PP, μ)
    end
    Vf = Vector{LinInterp}(2) 
    cf = Vector{LinInterp}(2) 
    nf = Vector{LinInterp}(2) 
    xprimef = Array{LinInterp}(2, S) 
    for s in 1:2  
        cf[s] = LinInterp(x[:, s][end:-1:1], c[:, s][end:-1:1])
```
nf[s] = LinInterp(x[:, s][end:-1:1], n[:, s][end:-1:1])
Vf[s] = LinInterp(x[:, s][end:-1:1], V[:, s][end:-1:1])
for sprime in 1:S
    xprimef[s, sprime] = LinInterp(x[:, s][end:-1:1], x[:, s][end:-1:1])
end
policies = [cf, nf, xprimef]
# Create xgrid
xbar = [maximum(minimum(x, 1)), minimum(maximum(x, 1))]
xgrid = linspace(xbar[1], xbar[2], length(grid0))
# Now iterate on bellman equation
T = BellmanEquation(model, xgrid, policies)
diff = 1.0
while diff > 1e-6
    if T.time_0 == false
        Vfnew, policies =
            fit_policy_function(PP,
                (i_x, x, s) -> get_policies_time1(T, i_x, x, s, Vf), xgrid)
    elseif T.time_0 == true
        Vfnew, policies =
            fit_policy_function(PP,
                (i_x, B_, s0) -> get_policies_time0(T, i_x, B_, s0, Vf),
                xgrid)
    else
        error("T.time_0 is $(T.time_0), which is invalid")
    end
    diff = 0.0
    for s in 1:S
        diff = max(diff, maximum(abs, (Vf[s](xgrid) - Vfnew[s].(xgrid)))).
    end
    print("diff = $diff \n")
    Vf = Vfnew
end
# Store value function policies and Bellman Equations
return Vf, policies, T, xgrid
end

""
Fits the policy functions PF using the points `xgrid` using interpolation
""
function fit_policy_function(PP::SequentialAllocation,
                               PF::Function, xgrid::AbstractArray)
    S = PP.S
    Vf = Vector(LinInterp)(S)
    cf = Vector(LinInterp)(S)
    nf = Vector(LinInterp)(S)
    xprimef = Array{LinInterp}(S, S)
    for s in 1:S
        PFvec = Array{typeof(PP.model).parameters[1]}(length(xgrid), 3+S)
        for (i_x, x) in enumerate(xgrid)
            PFvec[i_x, :) = PF(i_x, x, s)
        end
    end
end
end
Vf[s] = LinInterp(xgrid, PFvec[:, 1])
cf[s] = LinInterp(xgrid, PFvec[:, 2])
nf[s] = LinInterp(xgrid, PFvec[:, 3])
for sprime in 1:S
    xprimef[s, sprime] = LinInterp(xgrid, PFvec[:, 3+sprime])
end
end
return Vf, [cf, nf, xprimef]
end

""
Finds the optimal allocation given initial government debt `B_` and state `s_` → 0
""

function time0_allocation(pab::RecursiveAllocation, 
    B_::AbstractFloat, s0::Integer)
    xgrid = pab.xgrid
    if pab.T.time_0 == false
        z0 = get_policies_time1(pab.T, i_x, x, s, pab.Vf)
    elseif pab.T.time_0 == true
        z0 = get_policies_time0(pab.T, B_, s0, pab.Vf)
    else
        error("T.time_0 is $(T.time_0), which is invalid")
    end
    c0, n0, xprime0 = z0[2], z0[3], z0[4:end]
end

""
Simulates Ramsey plan for `T` periods
""

function simulate(pab::RecursiveAllocation, 
    B_::AbstractFloat, s_0::Integer, T::Integer, 
    sHist::Vector=QuantEcon.simulate(mc, s_0, T))
    model, S, policies = pab.model, pab.S, pab.policies
    β, Π, Uc = model.β, model.Π, model.Uc
    cf, nf, xprimef = policies[1], policies[2], policies[3]
    TF = typeof(model).parameters[1]
    cHist = Vector(TF)(T)
    nHist = Vector(TF)(T)
    BHist = Vector(TF)(T)
    THist = Vector(TF)(T)
    μHist = Vector(TF)(T)
    RHist = Vector(TF)(T-1)
    # time 0
    cHist[1], nHist[1], xprime = time0_allocation(pab, B_, s_0)
    THist[1] = T(pab.model, cHist[1], nHist[1])[s_0]
    BHist[1] = B_
    μHist[1] = 0.0
    # time 1 onward
    for t in 2:T
        s, x = sHist[t], xprime[sHist[t]]
        cHist[t], nHist[t], xprime = time0_allocation(pab, B_, s)
        THist[t] = T(pab.model, cHist[t], nHist[t])[s]
        BHist[t] = B_
        μHist[t] = 0.0
    end
end

6.3. Optimal Taxation with State-Contingent Debt
6.3.4 Examples

Anticipated One Period War

This example illustrates in a simple setting how a Ramsey planner manages risk.

Government expenditures are known for sure in all periods except one:

- For $t < 3$ and $t > 3$ we assume that $g_t = g_l = 0.1$.
- At $t = 3$ a war occurs with probability 0.5.
  - If there is war, $g_3 = g_h = 0.2$.
  - If there is no war $g_3 = g_l = 0.1$.

We define the components of the state vector as the following six $(t, g)$ pairs:

$(0, g_l), (1, g_l), (2, g_l), (3, g_l), (3, g_h), (t \geq 4, g_l)$.

We think of these 6 states as corresponding to $s = 1, 2, 3, 4, 5, 6$.

The transition matrix is

$$
\Pi = \begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.5 & 0.5 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 \\
\end{pmatrix}
$$

Government expenditures at each state are

$$
g = \begin{pmatrix}
0.1 \\
0.1 \\
0.1 \\
0.1 \\
0.2 \\
0.1 \\
\end{pmatrix}
$$
We assume that the representative agent has utility function

\[ u(c, n) = \frac{c^{1-\sigma} - n^{1+\gamma}}{1-\sigma} \]

and set \( \sigma = 2, \gamma = 2 \), and the discount factor \( \beta = 0.9 \).

Note: For convenience in terms of matching our code, we have expressed utility as a function of \( n \) rather than leisure \( l \).

This utility function is implemented in the type `CRRAutility`

```julia
function crra_utility(
    \( \beta = 0.9 \),
    \( \sigma = 2.0 \),
    \( \gamma = 2.0 \),
    \( \Pi = 0.5 \times \text{ones}(2, 2) \),
    \( G = [0.1, 0.2] \),
    \( \Theta = \text{ones}(\text{Float64}, 2) \),
    transfers = \text{false}
)

    function U(c, n)
        if \( \sigma == 1.0 \)
            U = log(c)
        else
            U = (c.(^\( 1.0 - \sigma \)) - 1.0) / (1.0 - \sigma)
        end
        return U - n.^\( 1 + \gamma \) / (1 + \gamma)
    end

    # Derivatives of utility function
    Uc(c, n) = c.^\(-\sigma\)
    Ucc(c, n) = -\sigma * c.^\(-\sigma - 1.0\)
    Un(c, n) = -n.^\gamma
    Unn(c, n) = -\gamma * n.^\( \gamma - 1.0 \)
    n_less_than_one = \text{false}
    return Model(\( \beta, \Pi, G, \Theta, \) transfers,
                    U, Uc, Ucc, Un, Unn, n_less_than_one)
end
```

We set initial government debt \( b_0 = 1 \).

We can now plot the Ramsey tax under both realizations of time \( t = 3 \) government expenditures

- black when \( g_3 = .1 \), and
- red when \( g_3 = .2 \)

```julia
M_time_example = crra_utility(G=[0.1, 0.1, 0.1, 0.2, 0.1, 0.1],
                             \( \Theta=\text{ones}(6) \)) # \( \Theta \) can in principle be random

M_time_example.\Pi = [0.0 1.0 0.0 0.0 0.0 0.0;
                      0.0 0.0 1.0 0.0 0.0 0.0;
                      0.0 0.0 0.0 0.5 0.5 0.0;
                      0.0 0.0 0.0 0.0 0.0 1.0;
                      0.0 0.0 0.0 0.0 0.0 1.0;]
```

6.3. Optimal Taxation with State-Contingent Debt
0.0 0.0 0.0 0.0 0.0 1.0

PP_seq_time = SequentialAllocation(M_time_example) # Solve sequential problem

sHist_h = [1, 2, 3, 4, 6, 6, 6]
sHist_l = [1, 2, 3, 5, 6, 6, 6]

sim_seq_h = simulate(PP_seq_time, 1.0, 1, 7, sHist_h)
sim_seq_l = simulate(PP_seq_time, 1.0, 1, 7, sHist_l)

using PyPlot
titles = hcat("Consumption",
    "Labor Supply",
    "Government Debt",
    "Tax Rate",
    "Government Spending",
    "Output")

sim_seq_l_plot = [sim_seq_l[1:4]..., M_time_example.G[sHist_l],
                  M_time_example.Θ[sHist_l] .* sim_seq_l[2]]
sim_seq_h_plot = [sim_seq_h[1:4]..., M_time_example.G[sHist_h],
                  M_time_example.Θ[sHist_h] .* sim_seq_h[2]]

plt[:figure](figsize=[14, 10])
for i = 1:6
    plt[:subplot](3, 2, i)
    plt[:title](titles[i])
    plt[:plot](sim_seq_l_plot[i], "-ok")
    plt[:plot](sim_seq_h_plot[i], "-or")
    grid("on")
end

plt[:tight_layout]()
Tax smoothing

- the tax rate is constant for all $t \geq 1$
  - For $t \geq 1, t \neq 3$, this is a consequence of $g_t$ being the same at all those dates
  - For $t = 3$, it is a consequence of the special one-period utility function that we have assumed
  - Under other one-period utility functions, the time $t = 3$ tax rate could be either higher or lower than for dates $t \geq 1, t \neq 3$
- the tax rate is the same at $t = 3$ for both the high $g_t$ outcome and the low $g_t$ outcome

We have assumed that at $t = 0$, the government owes positive debt $b_0$

It sets the time $t = 0$ tax rate partly with an eye to reducing the value $u_{c,0}b_0$ of $b_0$

It does this by increasing consumption at time $t = 0$ relative to consumption in later periods

This has the consequence of raising the time $t = 0$ value of the gross interest rate for risk-free loans between periods $t$ and $t + 1$, which equals

$$R_t = \frac{u_{c,t}}{\beta E_t[u_{c,t+1}]}$$

A tax policy that makes time $t = 0$ consumption be higher than time $t = 1$ consumption evidently increases the risk-free rate one-period interest rate, $R_t$, at $t = 0$
Raising the time $t = 0$ risk-free interest rate makes time $t = 0$ consumption goods cheaper relative to consumption goods at later dates, thereby lowering the value $u_{c,0}b_0$ of initial government debt $b_0$.

We see this in a figure below that plots the time path for the risk free interest rate under both realizations of the time $t = 3$ government expenditure shock.

The following plot illustrates how the government lowers the interest rate at time 0 by raising consumption.

```
plt[:figure](figsize=[8, 5])
plt[:title]("Gross Interest Rate")
plt[:plot](sim_seq_l[end], "-ok")
plt[:plot](sim_seq_h[end], "-or")
plt[:tight_layout]()
grid("on")
```

**Government Saving**

At time $t = 0$ the government evidently *dissaves* since $b_1 > b_0$.

- This is a consequence of it setting a lower tax rate at $t = 0$, implying more consumption at $t = 0$.

At time $t = 1$, the government evidently *saves* since it has set the tax rate sufficiently high to allow it to set $b_2 < b_1$.

- Its motive for doing this is that it anticipates a likely war at $t = 3$.

At time $t = 2$ the government trades state-contingent Arrow securities to hedge against war at $t = 3$. 

956 Chapter 6. Dynamic Programming Squared
• It purchases a security that pays off when \( g_3 = g_h \)
• It sells a security that pays off when \( g_3 = g_l \)
• These purchases are designed in such a way that regardless of whether or not there is a war at \( t = 3 \), the government will begin period \( t = 4 \) with the same government debt
• The time \( t = 4 \) debt level can be serviced with revenues from the constant tax rate set at times \( t \geq 1 \)

At times \( t \geq 4 \) the government rolls over its debt, knowing that the tax rate is set at level required to service the interest payments on the debt and government expenditures

**Time 0 Manipulation of Interest Rate**

We have seen that when \( b_0 > 0 \), the Ramsey plan sets the time \( t = 0 \) tax rate partly with an eye toward raising a risk-free interest rate for one-period loans between times \( t = 0 \) and \( t = 1 \)

By raising this interest rate, the plan makes time \( t = 0 \) goods cheap relative to consumption goods at later times

By doing this, it lowers the value of time \( t = 0 \) debt that it has inherited and must finance

**Time 0 and Time-Inconsistency**

In the preceding example, the Ramsey tax rate at time 0 differs from its value at time 1

To explore what is going on here, lets simplify things by removing the possibility of war at time \( t = 3 \)

The Ramsey problem then includes no randomness because \( g_t = g_l \) for all \( t \)

The figure below plots the Ramsey tax rates and gross interest rates at time \( t = 0 \) and time \( t = 1 \) as functions of the initial government debt (using the sequential allocation solution and a CRRA utility function defined above)

```plaintext
M2 = crra_utility(G=[0.15], \Pi=ones(1, 1), \Theta=[1.0])
PP_seq_time0 = SequentialAllocation(M2) # solve sequential problem
B_vec = linspace(-1.5, 1.0, 100)
taxpolicy = hcat([simulate(PP_seq_time0, B_, 1, 2) for B_ in B_vec]...)
interest_rate = hcat([simulate(PP_seq_time0, B_, 1, 3)[end] for B_ in B_~vec]...)

titles = ["Tax Rate", "Gross Interest Rate"]
plt[:figure](figsize=(10, 8))
for (i, series) in enumerate((taxpolicy, interest_rate))
    plt[:subplot](2, 1, i)
    plt[:plot](B_vec, series, linewidth = 2.0)
    plt[:title](titles[i])
    grid("on")
end
plt[:subplot](2, 1, 1)
plt[:legend]((latexstring("Time ", "t=0"),
```

6.3. Optimal Taxation with State-Contingent Debt
The figure indicates that if the government enters with positive debt, it sets a tax rate at $t = 0$ that is less than all later tax rates.

By setting a lower tax rate at $t = 0$, the government raises consumption, which reduces the value $u_{c,0}b_0$ of its initial debt.

It does this by increasing $c_0$ and thereby lowering $u_{c,0}$.

Conversely, if $b_0 < 0$, the Ramsey planner sets the tax rate at $t = 0$ higher than in subsequent periods.

A side effect of lowering time $t = 0$ consumption is that it raises the one-period interest rate at time 0 above that of subsequent periods.

There are only two values of initial government debt at which the tax rate is constant for all $t \geq 0$.

The first is $b_0 = 0$. 

---

**Chapter 6. Dynamic Programming Squared**
Here the government can use the \( t = 0 \) tax rate to alter the value of the initial debt.

The second occurs when the government enters with sufficiently large assets that the Ramsey planner can achieve first best and sets \( \tau_t = 0 \) for all \( t \).

It is only for these two values of initial government debt that the Ramsey plan is time-consistent.

Another way of saying this is that, except for these two values of initial government debt, a continuation of a Ramsey plan is not a Ramsey plan.

To illustrate this, consider a Ramsey planner who starts with an initial government debt \( b_1 \) associated with one of the Ramsey plans computed above.

Call \( \tau_1^R \) the time \( t = 0 \) tax rate chosen by the Ramsey planner confronting this value for initial government debt.

The figure below shows both the tax rate at time 1 chosen by our original Ramsey planner and what a new Ramsey planner would choose for its time \( t = 0 \) tax rate.

```julia
# Compute the debt entered with at time 1
B1_vec = hcat([simulate(PP_seq_time0, B_, 1, 2)[3][2] for B_ in B_vec])

# Compute the optimal policy if the government could reset
tau1_reset = hcat([simulate(PP_seq_time0, B1, 1, 1)[4] for B1 in B1_vec])

plt[:figure](figsize=[10, 6])
plt[:plot](B_vec, taxpolicy[:, 2], linewidth=2.)
plt[:plot](B_vec, tau1_reset, linewidth=2.)
plt[:xlabel]("Initial Government Debt")
plt[:title]("Tax Rate")
plt[:legend]((L"\tau_1", L"\tau_1^R"), loc=2, shadow=true)
plt[:tight_layout]()
grid("on")
```
The tax rates in the figure are equal for only two values of initial government debt.

**Tax Smoothing and non-CRRA Preferences**

The complete tax smoothing for \( t \geq 1 \) in the preceding example is a consequence of our having assumed CRRA preferences.

To see what is driving this outcome, we begin by noting that the Ramsey tax rate for \( t \geq 1 \) is a time invariant function \( \tau(\Phi, g) \) of the Lagrange multiplier on the implementability constraint and government expenditures.

For CRRA preferences, we can exploit the relations \( U_{cc} = -\sigma U_c \) and \( U_{nn} = \gamma U_n \) to derive

\[
\frac{(1 + (1 - \sigma)\Phi)U_c}{(1 + (1 - \gamma)\Phi)U_n} = 1
\]

from the first-order conditions.

This equation immediately implies that the tax rate is constant.

For other preferences, the tax rate may not be constant.

For example, let the period utility function be

\[ u(c, n) = \log(c) + 0.69 \log(1 - n) \]

We will create a new type `LogUtility` to represent this utility function.
Also suppose that \( g_t \) follows a two state i.i.d. process with equal probabilities attached to \( g_l \) and \( g_h \)

To compute the tax rate, we will use both the sequential and recursive approaches described above

The figure below plots a sample path of the Ramsey tax rate
As should be expected, the recursive and sequential solutions produce almost identical allocations.

Unlike outcomes with CRRA preferences, the tax rate is not perfectly smoothed.

Instead the government raises the tax rate when $g_t$ is high.

Further Comments

A related lecture describes an extension of the Lucas-Stokey model by Aiyagari, Marcet, Sargent, and Seppälä (2002) [AMSS02].

In the AMSS economy, only a risk-free bond is traded.

That lecture compares the recursive representation of the Lucas-Stokey model presented in this lecture with one for an AMSS economy.

By comparing these recursive formulations, we shall glean a sense in which the dimension of the state is lower in the Lucas Stokey model.
Accompanying that difference in dimension will be different dynamics of government debt

## 6.4 Optimal Taxation without State-Contingent Debt

### Contents

- *Optimal Taxation without State-Contingent Debt*
  - *Overview*
  - *Competitive Equilibrium with Distorting Taxes*
  - *Recursive Version of AMSS Model*
  - *Examples*

### 6.4.1 Overview

In *an earlier lecture* we described a model of optimal taxation with state-contingent debt due to Robert E. Lucas, Jr., and Nancy Stokey [*LS83*]

Aiyagari, Marcet, Sargent, and Seppälä [*AMSS02*] (hereafter, AMSS) studied optimal taxation in a model without state-contingent debt

In this lecture, we

- describe assumptions and equilibrium concepts
- solve the model
- implement the model numerically
- conduct some policy experiments
- compare outcomes with those in a corresponding complete-markets model

We begin with an introduction to the model

### 6.4.2 Competitive Equilibrium with Distorting Taxes

Many but not all features of the economy are identical to those of the Lucas-Stokey economy

Let's start with things that are identical

For $t \geq 0$, a history of the state is represented by $s^t = [s_t, s_{t-1}, \ldots, s_0]$

Government purchases $g(s)$ are an exact time-invariant function of $s$

Let $c_t(s^t)$, $\ell_t(s^t)$, and $n_t(s^t)$ denote consumption, leisure, and labor supply, respectively, at history $s^t$ at time $t$
Each period a representative household is endowed with one unit of time that can be divided between leisure $\ell_t$ and labor $n_t$:

$$n_t(s^t) + \ell_t(s^t) = 1$$  \hfill (6.98)

Output equals $n_t(s^t)$ and can be divided between consumption $c_t(s^t)$ and $g(s_t)$

$$c_t(s^t) + g(s_t) = n_t(s^t)$$  \hfill (6.99)

Output is not storable

The technology pins down a pre-tax wage rate to unity for all $t, s^t$

A representative household’s preferences over \{c_t(s^t), \ell_t(s^t)\}$_{t=0}^{\infty}$ are ordered by

$$\sum_{t=0}^{\infty} \sum_{s^t} \beta^t \pi_t(s^t) u[c_t(s^t), \ell_t(s^t)]$$  \hfill (6.100)

where

- $\pi_t(s^t)$ is a joint probability distribution over the sequence $s^t$, and
- the utility function $u$ is increasing, strictly concave, and three times continuously differentiable in both arguments

The government imposes a flat rate tax $\tau_t(s^t)$ on labor income at time $t$, history $s^t$

Lucas and Stokey assumed that there are complete markets in one-period Arrow securities; also see smoothing models

It is at this point that AMSS [AMSS02] modify the Lucas and Stokey economy

AMSS allow the government to issue only one-period risk-free debt each period

Ruling out complete markets in this way is a step in the direction of making total tax collections behave more like that prescribed in [Bar79] than they do in [LS83]

**Risk-free One-Period Debt Only**

In period $t$ and history $s^t$, let

- $b_{t+1}(s^t)$ be the amount of the time $t + 1$ consumption good that at time $t$ the government promised to pay
- $R_t(s^t)$ be the gross interest rate on risk-free one-period debt between periods $t$ and $t + 1$
- $T_t(s^t)$ be a nonnegative lump-sum transfer to the representative household\(^1\)

\(^1\) In an allocation that solves the Ramsey problem and that levies distorting taxes on labor, why would the government ever want to hand revenues back to the private sector? It would not in an economy with state-contingent debt, since any such allocation could be improved by lowering distortionary taxes rather than handing out lump-sum transfers. But without state-contingent debt there can be circumstances when a government would like to make lump-sum transfers to the private sector.
That $b_{t+1}(s^t)$ is the same for all realizations of $s_{t+1}$ captures its risk-free character.

The market value at time $t$ of government debt maturing at time $t + 1$ equals $b_{t+1}(s^t)$ divided by $R_t(s^t)$.

The government's budget constraint in period $t$ at history $s^t$ is

$$b_t(s^{t-1}) = \pi^n_t(s^t) n_t(s^t) - g_t(s_t) - T_t(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)}$$

$$\equiv z(s^t) + \frac{b_{t+1}(s^t)}{R_t(s^t)},$$

where $z(s^t)$ is the net-of-interest government surplus.

To rule out Ponzi schemes, we assume that the government is subject to a natural debt limit (to be discussed in a forthcoming lecture).

The consumption Euler equation for a representative household able to trade only one-period risk-free debt with one-period gross interest rate $R_t(s^t)$ is

$$\frac{1}{R_t(s^t)} = \sum_{s^{t+1}|s^t} \beta \pi_{t+1}(s^{t+1}|s^t) \frac{u_c(s^{t+1})}{u_c(s^t)}$$

Substituting this expression into the government's budget constraint (6.101) yields:

$$b_t(s^{t-1}) = z(s^t) + \beta \sum_{s^{t+1}|s^t} \pi_{t+1}(s^{t+1}|s^t) \frac{u_c(s^{t+1})}{u_c(s^t)} - \frac{b_{t+1}(s^t)}{R_{t+1}(s^{t+1})}$$

Components of $z(s^t)$ on the right side depend on $s^t$, but the left side is required to depend on $s^{t-1}$ only.

This is what it means for one-period government debt to be risk-free.

Therefore, the sum on the right side of equation (6.102) also has to depend only on $s^{t-1}$.

This requirement will give rise to measurability constraints on the Ramsey allocation to be discussed soon.

If we replace $b_{t+1}(s^t)$ on the right side of equation (6.102) by the right side of next periods budget constraint (associated with a particular realization $s_t$) we get

$$b_t(s^{t-1}) = z(s^t) + \sum_{s^{t+1}|s^t} \beta \pi_{t+1}(s^{t+1}|s^t) \frac{u_c(s^{t+1})}{u_c(s^t)} \left[ z(s^{t+1}) + \frac{b_{t+2}(s^{t+1})}{R_{t+2}(s^{t+1})} \right]$$

After making similar repeated substitutions for all future occurrences of government indebtedness, and by invoking the natural debt limit, we arrive at:

$$b_t(s^{t-1}) = \sum_{j=0}^{\infty} \sum_{s^{t+j}|s^t} \beta^j \pi_{t+j}(s^{t+j}|s^t) \frac{u_c(s^{t+j})}{u_c(s^t)} z(s^{t+j})$$

Now lets...
• substitute the resource constraint into the net-of-interest government surplus, and
• use the households first-order condition \( 1 - \tau_t^{n}(s^t) = \frac{u_t(s^t)}{u_c(s^t)} \) to eliminate the labor tax rate so that we can express the net-of-interest government surplus \( z(s^t) \) as

\[
z(s^t) = \left[ 1 - \frac{u_t(s^t)}{u_c(s^t)} \right] \left[ c_t(s^t) + g_t(s_t) \right] - g_t(s_t) - T_t(s^t). \tag{6.104}
\]

If we substitute the appropriate versions of right side of (6.104) for \( z(s^{t+j}) \) into equation (6.103), we obtain a sequence of implementability constraints on a Ramsey allocation in an AMSS economy:

Expression (6.103) at time \( t = 0 \) and initial state \( s^0 \) was also an implementability constraint on a Ramsey allocation in a Lucas-Stokey economy:

\[
b_0(s^{-1}) = \mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} z(s^j) \tag{6.105}
\]

Indeed, it was the only implementability constraint there.

But now we also have a large number of additional implementability constraints

\[
b_t(s^{t-1}) = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} z(s^{t+j}) \tag{6.106}
\]

Equation (6.106) must hold for each \( s^t \) for each \( t \geq 1 \)

### Comparison with Lucas-Stokey Economy

The expression on the right side of (6.106) in the Lucas-Stokey (1983) economy would equal the present value of a continuation stream of government surpluses evaluated at what would be competitive equilibrium Arrow-Debreu prices at date \( t \).

In the Lucas-Stokey economy, that present value is measurable with respect to \( s^t \).

In the AMSS economy, the restriction that government debt be risk-free imposes that that same present value must be measurable with respect to \( s^{t-1} \).

In a language used in the literature on incomplete markets models, it can be said that the AMSS model requires that at each \( (t, s^t) \) what would be the present value of continuation government surpluses in the Lucas-Stokey model must belong to the marketable subspace of the AMSS model.

### Ramsey Problem Without State-contingent Debt

After we have substituted the resource constraint into the utility function, we can express the Ramsey problem as being to choose an allocation that solves

\[
\max_{\{c_t(s^t), b_{t+1}(s^t)\}} \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u \left( c_t(s^t), 1 - c_t(s^t) - g_t(s_t) \right)
\]
where the maximization is subject to

\[
\mathbb{E}_0 \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^j)}{u_c(s^0)} z(s^j) \geq b_0(s^{-1}) \tag{6.107}
\]

and

\[
\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} z(s^{t+j}) = b_t(s^{t-1}) \quad \forall s^t \tag{6.108}
\]
given \(b_0(s^{-1})\)

### Lagrangian Formulation

Let \(\gamma_0(s^0)\) be a nonnegative Lagrange multiplier on constraint (6.107)

As in the Lucas-Stokey economy, this multiplier is strictly positive when the government must resort to distortionary taxation; otherwise it equals zero

A consequence of the assumption that there are no markets in state-contingent securities and that a market exists only in a risk-free security is that we have to attach stochastic processes \(f_t(s^t)\) of Lagrange multipliers to the implementability constraints (6.108)

Depending on how the constraints bind, these multipliers can be positive or negative:

\[
\gamma_t(s^t) \geq (\leq) 0 \quad \text{if the constraint binds in this direction}
\]

\[
\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j \frac{u_c(s^{t+j})}{u_c(s^t)} z(s^{t+j}) \geq (\leq) b_t(s^{t-1}).
\]

A negative multiplier \(\gamma_t(s^t) < 0\) means that if we could relax constraint (6.108), we would like to increase the beginning-of-period indebtedness for that particular realization of history \(s^t\)

That would let us reduce the beginning-of-period indebtedness for some other history\(^2\)

These features flow from the fact that the government cannot use state-contingent debt and therefore cannot allocate its indebtedness efficiently across future states

### Some Calculations

It is helpful to apply two transformations to the Lagrangian

Multiply constraint (6.107) by \(u_c(s^0)\) and the constraints (6.108) by \(\beta^j u_c(s^t)\)

Then a Lagrangian for the Ramsey problem can be represented as

\(^2\) From the first-order conditions for the Ramsey problem, there exists another realization \(\tilde{s}^t\) with the same history up until the previous period, i.e., \(\tilde{s}^{t-1} = s^{t-1}\), but where the multiplier on constraint (6.108) takes a positive value, so \(\gamma_t(\tilde{s}^t) > 0\).
\[ J = \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \left\{ u(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)) \right. \]
\[ \left. + \gamma_t(s^t) \left[ \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j u_c(s^{t+j}) z(s^{t+j}) - u_c(s^t) b_t(s^{t-1}) \right] \right\} \]
\[ = \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \left\{ u(c_t(s^t), 1 - c_t(s^t) - g_t(s_t)) \right. \]
\[ \left. + \Psi_t(s^t) u_c(s^t) z(s^t) - \gamma_t(s^t) u_c(s^t) b_t(s^{t-1}) \right\} \] (6.109)

where

\[ \Psi_t(s^t) = \Psi_{t-1}(s^{t-1}) + \gamma_t(s^t) \quad \text{and} \quad \Psi_{-1}(s^{-1}) = 0 \] (6.110)

In (6.109), the second equality uses the law of iterated expectations and Abels summation formula (also called summation by parts, see this page)

First-order conditions with respect to \( c_t(s^t) \) can be expressed as

\[ u_c(s^t) - u_c(s^t) + \Psi_t(s^t) \left\{ \left[ u_{cc}(s^t) - u_{ct}(s^t) \right] z(s^t) + u_c(s^t) z_c(s^t) \right\} \]
\[ - \gamma_t(s^t) \left[ u_{cc}(s^t) - u_{ct}(s^t) \right] b_t(s^{t-1}) = 0 \] (6.111)

and with respect to \( b_t(s^t) \) as

\[ \mathbb{E}_t \left[ \gamma_{t+1}(s^{t+1}) u_c(s^{t+1}) \right] = 0 \] (6.112)

If we substitute \( z(s^t) \) from (6.104) and its derivative \( z_c(s^t) \) into first-order condition (6.111), we find two differences from the corresponding condition for the optimal allocation in a Lucas-Stokey economy with state-contingent government debt

1. The term involving \( b_t(s^{t-1}) \) in first-order condition (6.111) does not appear in the corresponding expression for the Lucas-Stokey economy
   - This term reflects the constraint that beginning-of-period government indebtedness must be the same across all realizations of next periods state, a constraint that would not be present if government debt could be state contingent
2. The Lagrange multiplier \( \Psi_t(s^t) \) in first-order condition (6.111) may change over time in response to realizations of the state, while the multiplier \( \Phi \) in the Lucas-Stokey economy is time invariant

We need some code from our an earlier lecture on optimal taxation with state-contingent debt sequential allocation implementation:
using QuantEcon
using NLsolve
using NLopt

import QuantEcon.simulate

mutable struct Model{TF <: AbstractFloat,
    TM <: AbstractMatrix{TF},
    TV <: AbstractVector{TF}}
    β::TF
    II::TM
    G::TV
    Θ::TV
    transfers::Bool
    U::Function
    Uc::Function
    Ucc::Function
    Un::Function
    Unn::Function
    n_less_than_one::Bool
end

""
{{ class_word }} returns planner's allocation as a function of the multiplier on the implementability constraint μ
"

struct SequentialAllocation{TP <: Model,
    TI <: Integer,
    TV <: AbstractVector}
    model::TP
    mc::MarkovChain
    S::TI
    cFB::TV
    nFB::TV
    zFB::TV
end

""
Initializes the {{ class_word }} from the calibration model
"

function SequentialAllocation(model::Model)
    β, II, G, Θ = model.β, model.II, model.G, model.Θ
    mc = MarkovChain(II)
    S = size(II, 1)  # Number of states
    # Now find the first best allocation
    cFB, nFB, zFB = find_first_best(model, S, 1)

    return SequentialAllocation(model, mc, S, cFB, nFB, zFB, zFB)
end

""
Find the first best allocation
""
function find_first_best(model::Model, S::Integer, version::Integer)
    if version != 1 && version != 2
        throw(ArgumentError("version must be 1 or 2"))
    end

    β, Θ, Uc, Un, G, Π = model.β, model.Θ, model.Uc, model.Un, model.G, model.Π

    function res!(out, z)
        c = z[1:S]
        n = z[S+1:end]
        out[1:S] = Θ .* Uc(c, n) + Un(c, n)
        out[S+1:end] = Θ .* n - c - G
    end

    res = nlsolve(res!, 0.5 * ones(2 * S))
    if converged(res) == false
        error("Could not find first best")
    end

    if version == 1
        cFB = res.zero[1:S]
        nFB = res.zero[S+1:end]
        ΞFB = Uc(cFB, nFB)
        zFB = vcat(cFB, nFB, ΞFB)
        return cFB, nFB, ΞFB, zFB
    elseif version == 2
        cFB = res.zero[1:S]
        nFB = res.zero[S+1:end]
        IFB = Uc(cFB, nFB) .* cFB + Un(cFB, nFB) .* nFB
        xFB = \(\text{eye}(S) - β * Π, IFB)\)
        zFB = [vcat(cFB[s], xFB[s], xFB) for s in 1:S]
        return cFB, nFB, IFB, xFB, zFB
    end
end

""
Computes optimal allocation for time t 1 for a given μ
""

function time1_allocation(pas::SequentialAllocation, μ::Real)
    model, S = pas.model, pas.S
    Θ, β, Π, G, Uc, Ucc, Un, Unn = model.Θ, model.β, model.Π, model.G,
    model.Uc, model.Ucc, model.Un, model.Unn

    function FOC!(out, z::Vector)
        c = z[1:S]
        n = z[S+1:2S]
        Ξ = z[2S+1:end]
        out[1:S] = Uc(c, n) - μ * (Ucc(c, n) .* c + Uc(c, n)) - Ξ
        # FOC c
        out[S+1:2S] = Un(c, n) - μ * (Unn(c, n) .* n + Un(c, n)) + Θ .* Ξ
        # FOC n
        out[2S+1:end] = Θ .* n - c - G
        # Resource constraint
    end
end
return out

end

# Find the root of the FOC
res = nlsolve(FOC!, pas.zFB)
if res.f_converged == false
    error("Could not find LS allocation.")
end

z = res.zero

res = nlsolve(FOC!, [0.0, pas.cFB[s_0], pas.nFB[s_0], pas.ZFB[s_0]])
if res.f_converged == false
    error("Could not find time 0 LS allocation.")
end

return (res.zero...)

end

"""
Finds the optimal allocation given initial government debt `B_` and state `s_` ⇒ 0
"""
function time0_allocation(pas::SequentialAllocation,
                           B_::AbstractFloat, s_0::Integer)
    model = pas.model
    \( \Pi, \Theta, G, \beta = \text{model.}\Pi, \text{model.}\Theta, \text{model.}G, \text{model.}\beta \)
    \( \text{Uc, Un, Unn} = \text{model.Uc, model.Ucc, model.Un, model.Unn} \)

    # First order conditions of planner's problem
    function FOC!(out, z)
        \( \mu, c, n, \Xi = z[1:S], z[S+1:2S], z[2S+1:end] \)
        \( I = \text{Uc}(c, n) \odot c + \text{Un}(c, n) \odot n \)
        \( x = (\text{eye}(S) - \beta \odot \text{model.}\Pi, I) \)
        return c, n, x, \Xi
    end

    # Find root
    res = nlsolve(FOC!, [0.0, pas.cFB[s_0], pas.nFB[s_0], pas.ZFB[s_0]])
    if res.f_converged == false
        error("Could not find time 0 LS allocation.")
    end
    return (res.zero...)
end

"""
Find the value associated with multiplier \( \mu \)
"""
function time1_value(pas::SequentialAllocation, \( \mu :: \text{Real} \))
    model = pas.model
    \( c, n, x, \Xi = \text{time1\_allocation}(\text{pas, } \mu) \)
    \( U\_\text{val} = \text{model.U.}(c, n) \)
V = \( \text{eye}(\text{pas}.S) - \text{model.} \beta \cdot \text{model.} \Pi, \text{U_val} \)

return c, n, x, V
end

""
Computes T given `c`, `n`
""
function T(model::Model, c::Union{Real, Vector}, n::Union{Real, Vector})
    Uc, Un = model.Uc.(c, n), model.Un.(c, n)
    return 1 + Un ./ (model.Θ .* Uc)
end

""
Simulates planners policies for `T` periods
""
function simulate(pas::SequentialAllocation,
    B_::AbstractFloat, s_0::Integer,
    T::Integer,
    sHist::Union{Vector, Void}=nothing)
    model = pas.model
    Π, β, Uc = model.Π, model.β, model.Uc

    if sHist == nothing
        sHist = QuantEcon.simulate(pas.mc, T, init=s_0)
    end
    cHist = zeros(T)
    nHist = zeros(T)
    Bhist = zeros(T)
    THist = zeros(T)
    μHist = zeros(T)
    RHist = zeros(T-1)
    # time 0
    μ, cHist[1], nHist[1], _ = time0_allocation(pas, B_, s_0)
    THist[1] = T(pas.model, cHist[1], nHist[1])[s_0]
    Bhist[1] = B_
    μHist[1] = μ
    # time 1 onward
    for t in 2:T
        c, n, x, Ξ = time1_allocation(pas, μ)
        u_c = Uc(c, n)
        s = sHist[t]
        THist[t] = T(pas.model, c, n)[s]
        Eu_c = dot(Π[\text{Hist}[t-1],:], u_c)
        cHist[t], nHist[t], Bhist[t] = c[s], n[s], x[s] / u_c[s]
        RHist[t-1] = Uc(cHist[t-1], nHist[t-1]) / (β + Eu_c)
        μHist[t] = μ
    end
    return cHist, nHist, Bhist, THist, sHist, μHist, RHist
end

""
Bellman equation for the continuation of the Lucas-Stokey Problem
""
mutable struct BellmanEquation{TP <: Model, 
    TI <: Integer, 
    TV <: AbstractVector, 
    TM <: AbstractMatrix{TV}, 
    TVV <: AbstractVector{TV}}

model::TP
S::TI
xbar::TV
time_0::Bool
z0::TM
cFB::TV
nFB::TV
xFB::TV
zFB::TVV

end

""
Initializes the {{ class_word }} from the calibration `model` 
""
function BellmanEquation(model::Model, xgrid::AbstractVector, 
policies0::Vector)
    S = size(model.Π, 1)  # Number of states
    xbar = [minimum(xgrid), maximum(xgrid)]
    time_0 = false
    cf, nfb, xprimef = policies0
    z0 = [vcat(cf[s](x), n[s](x), [xprimef[s, sprime](x) for sprime in 1:S])
         for x in xgrid, s in 1:S]
    cFB, nFB, IFB, xFB, zFB = find_first_best(model, S, 2)
    return BellmanEquation(model, S, xbar, time_0, z0, cFB, nFB, xFB, zFB)
end

""
Finds the optimal policies 
""
function get_policies_time1(T::BellmanEquation, 
    i_x::Integer, x::AbstractFloat, 
    s::Integer, Vf::AbstractArray)
    model, S = T.model, T.S
    β, Θ, G, Π = model.β, model.Θ, model.G, model.Π
    U, Uc, Un = model.U, model.Uc, model.Un

    function objf(z::Vector, grad)
        c, xprime = z[1], z[2:end]
        n=c+G[s]
        Vprime = [Vf[sprime](xprime[sprime]) for sprime in 1:S]
        return -(U(c, n) + β * dot(Π[s, :], Vprime))
    end

    function cons(z::Vector, grad)
        c, xprime = z[1], z[2:end]
        n=c+G[s]
        return x - Uc(c, n) * c - Un(c, n) * n - β * dot(Π[s, :], xprime)
    end

end

6.4. Optimal Taxation without State-Contingent Debt
lb = vcat(0, T.xbar[1] * ones(S))
ub = vcat(1 - G[s], T.xbar[2] * ones(S))
opt = Opt(:LN_COBYLA, length(T.z0[i_x, s]) - 1)
min_objective!(opt, objf)
equality_constraint!(opt, cons)
lower_bounds!(opt, lb)
upper_bounds!(opt, ub)
maxeval!(opt, 300)
maxtime!(opt, 10)
init = vcat(T.z0[i_x, s][1], T.z0[i_x, s][3:end])
for (i, val) in enumerate(init)
    if val > ub[i]
        init[i] = ub[i]
    elseif val < lb[i]
        init[i] = lb[i]
    end
end
(minf, minx, ret) = optimize(opt, init)
T.z0[i_x, s] = vcat(minx[1], minx[1] + G[s], minx[2:end])
return vcat(-minf, T.z0[i_x, s])

""
Finds the optimal policies ""
function get_policies_time0(T::BellmanEquation,
    B::AbstractFloat, s0::Integer, Vf::Array)
    model, S = T.model, T.S
    β, Θ, G, Π = model.β, model.Θ, model.G, model.Π
    U, Uc, Un = model.U, model.Uc, model.Un

    function objf(z, grad)
        c, xprime = z[1], z[2:end]
        n = c + G[s0]
        Vprime = [Vf[sprime](xprime[sprime]) for sprime in 1:S]
        return -(U(c, n) + β * dot(Π[s0, :], Vprime))
    end

    function cons(z::Vector, grad)
        c, xprime = z[1], z[2:end]
        n = c + G[s0]
        return -Uc(c, n) * (c - B) - Un(c, n) * n - β * dot(Π[s0, :], xprime)
    end

    lb = vcat(0, T.xbar[1] * ones(S))
    ub = vcat(1 - G[s0], T.xbar[2] * ones(S))
    opt = Opt(:LN_COBYLA, length(T.zFB[s0]) - 1)
    min_objective!(opt, objf)
equality_constraint!(opt, cons)
lower_bounds!(opt, lb)
upper_bounds!(opt, ub)
maxeval!(opt, 300)
maxtime!(opt, 10)
init = vcat(T.zFB[s0][1], T.zFB[s0][3:end])
for (i, val) in enumerate(init)
    if val > ub[i]
        init[i] = ub[i]
    elseif val < lb[i]
        init[i] = lb[i]
    end
end
(minf, minx, ret) = optimize(opt, init)
T.zFB[s0] = vcat(minx[1], minx[1] + G[s0], minx[2:end])
return vcat(-minf, T.zFB[s0])
To analyze the AMSS model, we find it useful to adopt a recursive formulation using techniques like those in our lectures on dynamic Stackelberg models and optimal taxation with state-contingent debt.

### 6.4.3 Recursive Version of AMSS Model

We now describe a recursive formulation of the AMSS economy.

We have noted that from the point of view of the Ramsey planner, the restriction to one-period risk-free securities:

- leaves intact the single implementability constraint on allocations (6.105) from the Lucas-Stokey economy, but
- adds measurability constraints (6.103) on functions of tails of allocations at each time and history.

We now explore how these constraints alter Bellman equations for a time 0 Ramsey planner and for time \( t \geq 1 \), history \( s^t \) continuation Ramsey planners.

#### Recasting State Variables

In the AMSS setting, the government faces a sequence of budget constraints:

\[
\tau_t(s^t) n_t(s^t) + T_t(s^t) + b_{t+1}(s^t)/R_t(s^t) = g_t + b_t(s^{t-1})
\]

where \( R_t(s^t) \) is the gross risk-free rate of interest between \( t \) and \( t+1 \) at history \( s^t \) and \( T_t(s^t) \) are nonnegative transfers.

Throughout this lecture, we shall set transfers to zero (for some issues about the limiting behavior of debt, this makes a possibly important difference from AMSS [AMSS02], who restricted transfers to be nonnegative).

In this case, the household faces a sequence of budget constraints:

\[
b_t(s^{t-1}) + (1 - \tau_t(s^t)) n_t(s^t) = c_t(s^t) + b_{t+1}(s^t)/R_t(s^t)
\]

The households first-order conditions are \( u_{c,t} = \beta R_t \mathbb{E}_t u_{c,t+1} + (1 - \tau_t) u_{c,t} = u_{l,t} \).

Using these to eliminate \( R_t \) and \( \tau_t \) from budget constraint (6.113) gives:

\[
b_t(s^{t-1}) + \frac{u_{l,t}(s^t)}{u_{c,t}(s^t)} n_t(s^t) = c_t(s^t) + \beta (\mathbb{E}_t u_{c,t+1}) b_{t+1}(s^t) / u_{c,t}(s^t)
\]

### 6.4. Optimal Taxation without State-Contingent Debt

975
or

\[ u_{c,t}(s^t)b_t(s^{t-1}) + u_{t,t}(s^t)n_t(s^t) = u_{c,t}(s^t)c_t(s^t) + \beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t) \]  

(6.115)

Now define

\[ x_t \equiv \beta b_{t+1}(s^t) \mathbb{E}_t u_{c,t+1} = u_{c,t}(s^t) \frac{b_{t+1}(s^t)}{R_t(s^t)} \]  

(6.116)

and represent the households budget constraint at time \( t \), history \( s^t \) as

\[ \frac{u_{c,t}x_{t-1}}{\beta \mathbb{E}_{t-1} u_{c,t}} = u_{c,t}c_t - u_{t,t}n_t + x_t \]  

(6.117)

for \( t \geq 1 \)

**Measurability Constraints**

Write equation (6.115) as

\[ b_t(s^{t-1}) = c_t(s^t) - \frac{u_{t,t}(s^t)}{u_{c,t}(s^t)} n_t(s^t) + \frac{\beta(\mathbb{E}_t u_{c,t+1})b_{t+1}(s^t)}{u_{c,t}} \]  

(6.118)

The right side of equation (6.118) expresses the time \( t \) value of government debt in terms of a linear combination of terms whose individual components are measurable with respect to \( s^t \)

The sum of terms on the right side of equation (6.118) must equal \( b_t(s^{t-1}) \)

That implies that it is has to be *measurable* with respect to \( s^{t-1} \)

Equations (6.118) are the **measurability constraints** that the AMSS model adds to the single time 0 implementation constraint imposed in the Lucas and Stokey model

**Two Bellman Equations**

Let \( \Pi(s|s_-) \) be a Markov transition matrix whose entries tell probabilities of moving from state \( s_- \) to state \( s \) in one period

Let

- \( V(x_-, s_-) \) be the continuation value of a continuation Ramsey plan at \( x_{t-1} = x_-, s_{t-1} = s_- \) for \( t \geq 1 \)
- \( W(b, s) \) be the value of the Ramsey plan at time 0 at \( b_0 = b \) and \( s_0 = s \)
We distinguish between two types of planners:

For \( t \geq 1 \), the value function for a **continuation Ramsey planner** satisfies the Bellman equation

\[
V(x_-, s_-) = \max_{\{n(s), x(s)\}} \sum_s \Pi(s|s_-) [u(n(s) - g(s), 1 - n(s)) + \beta V(x(s), s)]
\]  

subject to the following collection of implementability constraints, one for each \( s \in \mathcal{S} \):

\[
\frac{u_c(s)x_-}{\beta \sum \Pi(\tilde{s}|s_-)u_c(\tilde{s})} = u_c(s)(n(s) - g(s)) - u_t(s)n(s) + x(s)
\]  

A continuation Ramsey planner at \( t \geq 1 \) takes \((x_{t-1}, s_{t-1}) = (x_-, s_-)\) as given and before \( s \) is realized chooses \((n_t(s_t), x_t(s_t)) = (n(s), x(s))\) for \( s \in \mathcal{S} \).

The **Ramsey planner** takes \((b_0, s_0)\) as given and chooses \((n_0, x_0)\).

The value function \( W(b_0, s_0) \) for the time \( t = 0 \) Ramsey planner satisfies the Bellman equation

\[
W(b_0, s_0) = \max_{n_0, x_0} u(n_0 - g_0, 1 - n_0) + \beta V(x_0, s_0)
\]

where maximization is subject to

\[
u_{c,0}b_0 = u_{c,0}(n_0 - g_0) - u_{t,0}n_0 + x_0
\]

**Martingale Supercedes State-Variable Degeneracy**

Let \( \mu(s|s_-)\Pi(s|s_-) \) be a Lagrange multiplier on constraint (6.120) for state \( s \).

After forming an appropriate Lagrangian, we find that the continuation Ramsey planners first-order condition with respect to \( x(s) \) is

\[
\beta V_x(x(s), s) = \mu(s|s_-)
\]

Applying the envelope theorem to Bellman equation (6.119) gives

\[
V_x(x_-, s_-) = \sum_s \Pi(s|s_-) \mu(s|s_-) \frac{u_c(s)}{\beta \sum \Pi(\tilde{s}|s_-)u_c(\tilde{s})}
\]

Equations (6.123) and (6.124) imply that

\[
V_x(x_-, s_-) = \sum_s \left( \frac{u_c(s)}{\sum \Pi(\tilde{s}|s_-)u_c(\tilde{s})} \right) V_x(x(s), s)
\]
Equation (6.125) states that $V_x(x, s)$ is a risk-adjusted martingale.

Saying that $V_x(x, s)$ is a risk-adjusted martingale means that $V_x(x, s)$ is a martingale with respect to the probability distribution over $s^t$ sequences that is generated by the twisted transition probability matrix:

$$\tilde{\Pi}(s|s_-) = \frac{\Pi(s|s_-)u_c(s)}{\sum_s \Pi(s|s_-)u_c(s)}$$

**Exercise:** Please verify that $\tilde{\Pi}(s|s_-)$ is a valid Markov transition density, i.e., that its elements are all nonnegative and that for each $s_-$, the sum over $s$ equals unity.

### Absence of State Variable Degeneracy

Along a Ramsey plan, the state variable $x_t = x_t(s^t, b_0)$ becomes a function of the history $s^t$ and initial government debt $b_0$.

In *Lucas-Stokey model*, we found that

- a counterpart to $V_x(x, s)$ is time invariant and equal to the Lagrange multiplier on the Lucas-Stokey implementability constraint
- time invariance of $V_x(x, s)$ is the source of a key feature of the Lucas-Stokey model, namely, state variable degeneracy (i.e., $x_t$ is an exact function of $s_t$)

That $V_x(x, s)$ varies over time according to a twisted martingale means that there is no state-variable degeneracy in the AMSS model.

In the AMSS model, both $x$ and $s$ are needed to describe the state.

This property of the AMSS model transmits a twisted martingale component to consumption, employment, and the tax rate.

### Digression on Nonnegative Transfers

Throughout this lecture we have imposed that transfers $T_t = 0$.

AMSS [*AMSS02*] instead imposed a nonnegativity constraint $T_t \geq 0$ on transfers.

They also considered a special case of quasi-linear preferences, $u(c, l) = c + H(l)$

In this case, $V_x(x, s) \leq 0$ is a non-positive martingale.

By the martingale convergence theorem $V_x(x, s)$ converges almost surely.

Furthermore, when the Markov chain $\Pi(s|s_-)$ and the government expenditure function $g(s)$ are such that $g_t$ is perpetually random, $V_x(x, s)$ almost surely converges to zero.

For quasi-linear preferences, the first-order condition with respect to $n(s)$ becomes

$$(1 - \mu(s|s_-))(1 - w_l(s)) + \mu(s|s_-)n(s)u_H(s) = 0$$

When $\mu(s|s_-) = \beta V_x(x(s), x)$ converges to zero, in the limit $w_l(s) = 1 = u_c(s)$, so that $\tau(x(s), s) = 0$.

Thus, in the limit, if $g_t$ is perpetually random, the government accumulates sufficient assets to finance all expenditures from earnings on those assets, returning any excess revenues to the household as nonnegative lump sum transfers.
The recursive formulation is implemented as follows

```julia
using Dierckx

mutable struct BellmanEquation_Recursive{TP <: Model, TI <: Integer, TR <: Real}
    model::TP
    S::TI
    xbar::Array{TR}
    time_0::Bool
    z0::Array[Array]
    cFB::Vector{TR}
    nFB::Vector{TR}
    xFB::Vector{TR}
    zFB::Vector{Vector{TR}}
end

mutable struct RecursiveAllocation{TP <: Model, TI <: Integer, TVg <: AbstractVector, TT <: Tuple}
    model::TP
    mc::MarkovChain
    S::TI
    T::BellmanEquation_Recursive
    μgrid::TVg
    xgrid::TVg
    Vf::Array
    policies::TT
end

function RecursiveAllocation(model::Model, μgrid::AbstractArray)
    G = model.G
    S = size(model.Π, 1) # number of states
    mc = MarkovChain(model.Π)
    # now find the first best allocation
    Vf, policies, T, xgrid = solve_timel_bellman(model, μgrid)
    T.time_0 = true # Bellman equation now solves time 0
    return RecursiveAllocation(model, mc, S, T, μgrid, xgrid, Vf, policies)
end
```

6.4. Optimal Taxation without State-Contingent Debt
Solve the time 1 Bellman equation for calibration Model and initial grid $\mu$grid

```julia
function solve_time1_bellman{TR <: Real}(model::Model{TR}, μgrid::AbstractArray)
    Π = model.Π
    S = size(model.Π, 1)

    # First get initial fit from lucas stockey solution.
    # Need to change things to be ex_ante
    PP = SequentialAllocation(model)

    function incomplete_allocation(PP::SequentialAllocation, μ_, s_::Integer)
        c, n, x, V = time1_value(PP, μ_, s_)
        return c, n, dot(Π[s_, :], x), dot(Π[s_, :], V)
    end

    cf = Array{Function}(S, S)
    nf = Array{Function}(S, S)
    xprimef = Array{Function}(S, S)
    Vf = Vector{Function}(S)
    xgrid = Array{TR}(S, length(μgrid))

    for s_ in 1:S
        c = Array{TR}(length(μgrid), S)
        n = Array{TR}(length(μgrid), S)
        x = Array{TR}(length(μgrid))
        V = Array{TR}(length(μgrid))
        for (i_μ, μ) in enumerate(μgrid)
            c[i_μ, :] = incomplete_allocation(PP, μ, s_)
        end
        xprimes = repmat(x, 1, S)
        xgrid[s_, :) = x
        for sprime = 1:S
            splc = Spline1D(x[end:-1:1], c[:, sprime][end:-1:1], k=3)
            spln = Spline1D(x[end:-1:1], n[:, sprime][end:-1:1], k=3)
            splx = Spline1D(x[end:-1:1], xprimes[:, sprime][end:-1:1], k=3)
            cf[s_, sprime] = y -> splc(y)
            nf[s_, sprime] = y -> spln(y)
            xprimef[s_, sprime] = y -> splx(y)
            # cf[s_, sprime] = LinInterp(x[end:-1:1], c[:, sprime][end:-1:1])
            # nf[s_, sprime] = LinInterp(x[end:-1:1], n[:, sprime][end:-1:1])
            # xprimef[s_, sprime] = LinInterp(x[end:-1:1], xprimes[:, sprime][end:-1:1])
            # splV = Spline1D(x[end:-1:1], V[end:-1:1], k=3)
            Vf[s_] = splV(y)
            # Vf[s_] = LinInterp(x[end:-1:1], V[end:-1:1])
        end
    end
```

Chapter 6. Dynamic Programming Squared
end

policies = [cf, nf, xprimef]

# Create xgrid
xbar = [maximum(minimum(xgrid)), minimum(maximum(xgrid))]
xgrid = linspace(xbar[1], xbar[2], length(μgrid))

# Now iterate on Bellman equation
T = BellmanEquation_Recursive(model, xgrid, policies)
diff = 1.0
while diff > 1e-4
    PF = (i_x, x, s) -> get_policies_time1(T, i_x, x, s, Vf, xbar)
    Vfnew, policies = fit_policy_function(T, PF, xgrid)
    diff = 0.0
    for s=1:S
        diff = max(diff, maximum(abs, (Vf[s].(xgrid) - Vfnew[s].(xgrid)) .→ Vf[s].(xgrid)))
    end
    println("diff = ", round(diff, 4))
    Vf = copy(Vfnew)
end

return Vf, policies, T, xgrid
end

"""
Fits the policy functions
""

function fit_policy_function(T::BellmanEquation_Recursive, PF::Function, xgrid::AbstractVector{TF})

    S = T.S
    # preallocation
    PFvec = Array{T,1}(4S + 1, length(xgrid))
    cf = Array{Function}(S, S)
    nf = Array{Function}(S, S)
    xprimef = Array{Function}(S, S)
    TTf = Array{Function}(S, S)
    Vf = Vector{Function}(S)
    # fit policy functions
    for s_in 1:S
        for (i_x, x) in enumerate(xgrid)
            PFvec[:, i_x] = PF(i_x, x, s)
        end
        splV = Spline1D(xgrid, PFvec[1,:], k=3)
        Vf[s] = y -> splV(y)
        # Vf[s] = LinInterp(xgrid, PFvec[1,:])
        for spprime=1:S
            splc = Spline1D(xgrid, PFvec[1 + spprime,:], k=3)
            spln = Spline1D(xgrid, PFvec[1 + S + spprime,:], k=3)

6.4. Optimal Taxation without State-Contingent Debt
splxprime = Spline1D(xgrid, PFvec[1 + 2S + sprime, :], k=3)
splTT = Spline1D(xgrid, PFvec[1 + 3S + sprime, :], k=3)
cf[s_, sprime] = y -> splc(y)
nf[s_, sprime] = y -> spln(y)
xprimef[s_, sprime] = y -> splxprime(y)
TTf[s_, sprime] = y -> splTT(y)

end

end

policies = (cf, nf, xprimef, TTf)
return Vf, policies

end

""
Computes Tau given c,n
""

function Tau(pab::RecursiveAllocation,
c::AbstractArray,
n::AbstractArray)
model = pab.model
Uc, Un = model.Uc(c, n), model.Un(c, n)
return 1 + Un ./ (model.Θ .* Uc)
end

Tau(pab::RecursiveAllocation, c::Real, n::Real) = Tau(pab, [c], [n])

""
Finds the optimal allocation given initial government debt B_ and state s_0
""

function time0_allocation(pab::RecursiveAllocation, B_::Real, s_0::Integer)
    T, Vf = pab.T, pab.Vf
    xbar = T.xbar
    z0 = get_policies_time0(T, B_, s0, Vf, xbar)

    c0, n0, xprime0, T0 = z0[2], z0[3], z0[4], z0[5]
    return c0, n0, xprime0, T0
end

""
Simulates planners policies for `T` periods
""

function simulate{TF <: AbstractFloat}(pab::RecursiveAllocation,
    B_::TF, s_0::Integer, T::Integer,
    sHist::Vector=simulate(pab.mc, T, init=s_0))
    model, mc, Vf, S = pab.model, pab.mc, pab.Vf, pab.S
    Π, Uc = model.Π, model.Uc
    cf, nf, xprimef, TTf = pab.policies

    cHist = Array{TF}(T)
nHist = Array{TF}(T)
Bhist = Array{TF}(T)
xHist = Array{TF}(T)
TauHist = Array{TF}(T)
```julia
THist = Array{TF}(T)
μHist = Array{TF}(T)

# time 0
chist[1], nHist[1], xHist[1], THist[1] = time0_allocation(pab, B_, s_0)
TauHist[1] = Tau(pab, chist[1], nHist[1])[s_0]
BHist[1] = B_
μHist[1] = Vf[s_0](xHist[1])

# time 1 onward
for t in 2:T
    s_, x, s = sHist[t-1], xHist[t-1], sHist[t]
c = Array{TF}(S)
n = Array{TF}(S)
xprime = Array{TF}(S)
TT = Array{TF}(S)
    for sprime in 1:S
        c[sprime], n[sprime], xprime[sprime], TT[sprime] =
            cf[s_, sprime](x), nf[s_, sprime](x),
            xprimef[s_, sprime](x), TTf[s_, sprime](x)
    end
    Tau_val = Tau(pab, c, n)[s]
    u_c = Uc(c, n)
    Eu_c = dot(P[s_, :], u_c)
    μHist[t] = Vf[s](xprime[s])
    chist[t], nHist[t], BHist[t], TauHist[t] = c[s], n[s], x/Eu_c, Tau_val
    xHist[t], THist[t] = xprime[s], TT[s]
end
```

```
function BellmanEquation_Recursive{TF <: AbstractFloat}{model::Model{TF}, xgrid::AbstractVector}
    policies0::Array

    S = size(model.Π, 1)
    xbar = [minimum(xgrid), maximum(xgrid)]
    time_0 = false
    z0 = Array{Array}(length(xgrid), S)
cf, nf, xprimef = policies0[1], policies0[2], policies0[3]
for s in 1:S
    for (i_x, x) in enumerate(xgrid)
        cs = Array{TF}(S)
n = Array{TF}(S)
xprimes = Array{TF}(S)
            for j = 1:S
```

6.4. Optimal Taxation without State-Contingent Debt

983
cs[j], ns[j], xprimes[j] = cf[s, j](x), nf[s, j](x),
xprimef[s, j](x)
end

z0[i_x, s] = vcat(cs, ns, xprimes, zeros(S))
end

cFB, nFB, IFB, xFB, zFB = find_first_best(model, S, 2)
return BellmanEquation_Recursive(model, S, xbar, time_0, z0, cFB, nFB,)
end

### Finds the optimal policies

```
function get_policies_time1(T::BellmanEquation_Recursive,
i_x::Integer,
x::Real,
s::Integer,
Vf::AbstractArray{Function},
xbar::AbstractVector)
    model, S = T.model, T.S
    β, Θ, G, Π = model.β, model.Θ, model.G, model.Π
    U,Uc,Un = model.U, model.Uc, model.Un

    S_possible = sum(Π[s, :] > 0)
    sprimei_possible = find(Π[s, :] > 0)

    function objf(z, grad)
        c, xprime = z[1:S_possible], z[S_possible+1:2S_possible]
        n = (c + G[sprimei_possible]) ./ Θ[sprimei_possible]
        Vprime = [Vf[sprimei_possible[si]][xprime[si]] for si in 1:S_possible]
        return -dot(Π[s, sprimei_possible], U.(c, n) + β * Vprime)
    end

    function cons(out, z, grad)
        c, xprime, TT = z[1:S_possible], z[S_possible + 1:2S_possible], z[2S_possible +
        1:3S_possible]
        n = (c+G[sprimei_possible]) ./ Θ[sprimei_possible]
        u_c = Uc.(c, n)
        Eu_c = dot(Π[s, sprimei_possible], u_c)
        out .*= x .* u_c / Eu_c - u_c .* (c - TT) - Un(c, n) .* n - β * xprime
    end

    function cons_no_trans(out, z, grad)
        c, xprime = z[1:S_possible], z[S_possible + 1:2S_possible]
        n = (c + G[sprimei_possible]) ./ Θ[sprimei_possible]
        u_c = Uc.(c, n)
        Eu_c = dot(Π[s, sprimei_possible], u_c)
        out .*= x .* u_c / Eu_c - u_c .* (c - Un(c, n) .* n - β * xprime
    end

    if model.transfers == true
\[
\begin{align*}
\text{lb} &= \text{vcat}(\text{zeros}(S_{\text{possible}}), \text{ones}(S_{\text{possible}}) \times \text{xbar}[1], \text{zeros}(S_{\text{possible}})) \\
\text{if} & \quad \text{model.n_less_than_one} == \text{true} \\
\text{ub} &= \text{vcat}(\text{ones}(S_{\text{possible}}) - G[\text{sprimei_possible}], \\
& \quad \text{ones}(S_{\text{possible}}) \times \text{xbar}[2], \text{ones}(S_{\text{possible}})) \\
\text{else} \\
\text{ub} &= \text{vcat}(100 + \text{ones}(S_{\text{possible}}), \\
& \quad \text{ones}(S_{\text{possible}}) \times \text{xbar}[2], \\
& \quad 100 + \text{ones}(S_{\text{possible}})) \\
\text{end} \\
\text{init} &= \text{vcat}(T.\text{z0}[i_x, s_][\text{sprimei_possible}], \\
& \quad T.\text{z0}[i_x, s_][2S + \text{sprimei_possible}], \\
& \quad T.\text{z0}[i_x, s_][3S + \text{sprimei_possible}]) \\
\text{opt} &= \text{Opt}(:\text{LN_COBYLA}, 3S_{\text{possible}}) \\
& \quad \text{equality_constraint!}(\text{opt, cons, zeros}(S_{\text{possible}})) \\
\text{else} \\
\text{lb} &= \text{vcat}(\text{zeros}(S_{\text{possible}}), \text{ones}(S_{\text{possible}}) \times \text{xbar}[1]) \\
\text{if} & \quad \text{model.n_less_than_one} == \text{true} \\
\text{ub} &= \text{vcat}(\text{ones}(S_{\text{possible}}) - G[\text{sprimei_possible}], \text{ones}(S_{\text{possible}}) \times \text{xbar}[2]) \\
\text{else} \\
\text{ub} &= \text{vcat}(\text{ones}(S_{\text{possible}}), \text{ones}(S_{\text{possible}}) \times \text{xbar}[2]) \\
\text{end} \\
\text{init} &= \text{vcat}(T.\text{z0}[i_x, s_][\text{sprimei_possible}], \\
& \quad T.\text{z0}[i_x, s_][2S + \text{sprimei_possible}]) \\
\text{opt} &= \text{Opt}(:\text{LN_COBYLA}, 2S_{\text{possible}}) \\
& \quad \text{equality_constraint!}(\text{opt, cons_no_trans, zeros}(S_{\text{possible}})) \\
\text{end} \\
\text{init} > \text{ub} &= \text{ub}[	ext{init} > \text{ub}] \\
\text{init} < \text{lb} &= \text{lb}[	ext{init} < \text{lb}] \\
\text{min_objective!}(\text{opt, objf}) \\
\text{lower_bounds!}(\text{opt, lb}) \\
\text{upper_bounds!}(\text{opt, ub}) \\
\text{maxeval!}(\text{opt, 10000000}) \\
\text{maxtime!}(\text{opt, 10}) \\
\text{ftol_rel!}(\text{opt, 1e-8}) \\
\text{ftol_abs!}(\text{opt, 1e-8}) \\
\text{(minf, minx, ret)} &= \text{optimize}(\text{opt, init}) \\
\text{if} & \quad \text{ret} != :\text{SUCCESS} \land \text{ret} != :\text{ROUNDOFF_LIMITED} \land \text{ret} != :\text{MAXEVAL_REACHED} \\
& \quad \land \text{ret} != :\text{FTOL_REACHED} \land \text{ret} != :\text{MAXTIME_REACHED} \\
\text{error}(EXIT: \text{optimization failed: ret = $ret"}) \\
\text{end} \\
T.\text{z0}[i_x, s_][\text{sprimei_possible}] &= \text{minx}[1:S_{\text{possible}}] \\
T.\text{z0}[i_x, s_][S + \text{sprimei_possible}] &= \text{minx}[1:S_{\text{possible}}] + G[\text{sprimei_possible}] \\
T.\text{z0}[i_x, s_][2S + \text{sprimei_possible}] &= \text{minx}[S_{\text{possible}} + 1:2S_{\text{possible}}] \\
\text{if} & \quad \text{model.transfers} == \text{true} \\
T.\text{z0}[i_x, s_][3S + \text{sprimei_possible}] &= \text{minx}[2S_{\text{possible}} + 1:3S_{\text{possible}}]
\end{align*}
\]
else
    T.z0[i_x, s_][3S + spritei_possible] = zeros(S)
end

return vcat(-minf, T.z0[i_x, s_])
end

""
Finds the optimal policies
"
function get_policies_time0(T::BellmanEquation_Recursive,
    B::Real,
    s0::Integer,
    Vf::AbstractArray{Function},
    xbar::AbstractVector)

    model = T.model
    β, Θ, G = model.β, model.Θ, model.G
    U, Uc, Un = model.U, model.Uc, model.Un

    function objf(z, grad)
        c, xprime = z[1], z[2]
        n = (c + G[s0]) / Θ[s0]
        return -(U(c, n) + β * Vf[s0](xprime))
    end

    function cons(z, grad)
        c, xprime, TT = z[1], z[2], z[3]
        n = (c + G[s0]) / Θ[s0]
        return -Uc(c, n) * (c - B - TT) - Un(c, n) * n - β * xprime
    end

    cons_no_trans(z, grad) = cons(vcat(z, 0), grad)

    if model.transfers == true
        lb = [0.0, xbar[1], 0.0]
        if model.n_less_than_one == true
            ub = [1 - G[s0], xbar[2], 100]
        else
            ub = [100.0, xbar[2], 100.0]
        end
        init = vcat(T.zFB[s0][1], T.zFB[s0][3], T.zFB[s0][4])
        init = [0.95124922, -1.15926816, 0.0]
        opt = Opt(:LN_COBYLA, 3)
        equality_constraint!(opt, cons)
    else
        lb = [0.0, xbar[1]]
        if model.n_less_than_one == true
            ub = [1 - G[s0], xbar[2]]
        else
            ub = [100, xbar[2]]
        end
        init = vcat(T.zFB[s0][1], T.zFB[s0][3])
        init = [0.95124922, -1.15926816]
        opt = Opt(:LN_COBYLA, 2)
6.4.4 Examples

We now turn to some examples

Anticipated One-Period War

In our lecture on optimal taxation with state contingent debt we studied how the government manages uncertainty in a simple setting.

As in that lecture, we assume the one-period utility function

$$u(c, n) = \frac{c^{1-\sigma}}{1-\sigma} - \frac{n^{1+\gamma}}{1+\gamma}$$

Note: For convenience in matching our computer code, we have expressed utility as a function of $n$ rather than leisure $l$

We consider the same government expenditure process studied in the lecture on optimal taxation with state contingent debt

Government expenditures are known for sure in all periods except one

- For $t < 3$ or $t > 3$ we assume that $g_t = g_t = 0.1$
• At $t = 3$ a war occurs with probability 0.5
  – If there is war, $g_3 = g_h = 0.2$
  – If there is no war $g_3 = g_l = 0.1$

A useful trick is to define components of the state vector as the following six $(t, g)$ pairs:

$$(0, g_l), (1, g_l), (2, g_l), (3, g_l), (3, g_h), (t \geq 4, g_l)$$

We think of these 6 states as corresponding to $s = 1, 2, 3, 4, 5, 6$

The transition matrix is

$$P = \begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.5 & 0.5 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1
\end{pmatrix}$$

The government expenditure at each state is

$$g = \begin{pmatrix}
0.1 \\
0.1 \\
0.1 \\
0.1 \\
0.2 \\
0.1
\end{pmatrix}$$

We assume the same utility parameters as in the *Lucas-Stokey economy*

This utility function is implemented in the following type

```julia
function crra_utility();
    \( \beta = 0.9, \)
    \( \sigma = 2.0, \)
    \( \gamma = 2.0, \)
    \( \Pi = 0.5 * \text{ones}(2, 2), \)
    \( G = [0.1, 0.2], \)
    \( \Theta = \text{ones}(\text{Float64}, 2), \)
    \( \text{transfers} = \text{false} \)

    function U(c, n)
        if \( \sigma == 1.0 \)
            U = log(c)
        else
            U = (c.^((1.0 - \sigma) - 1.0) / (1.0 - \sigma))
        end
        return U - n.^((1 + \gamma) / (1 + \gamma))
    end

    # Derivatives of utility function
    Uc(c, n) = c.^(-\sigma)
    Ucc(c, n) = -\sigma * c.^(-\sigma - 1.0)
```
\begin{align*}
Un(c,n) &= -n.\gamma \\
Unn(c,n) &= -\gamma \ast n.\gamma (\gamma - 1.0) \\
n_{\text{less than one}} &= \text{false} \\
\text{return Model(}\beta, \Pi, G, \Theta, \text{transfers,} \\
U, Uc, Ucc, Un, Unn, n_{\text{less than one}}) \\
\end{align*}

The following figure plots the Ramsey plan under both complete and incomplete markets for both possible realizations of the state at time \( t = 3 \).

Optimal policies when the government has access to state contingent debt are represented by black lines, while the optimal policies when there is only a risk free bond are in red.

Paths with circles are histories in which there is peace, while those with triangle denote war.

```julian
using Plots
pyplot()
titles = hcat("Consumption", "Labor", "Government Debt", 
"Tax Rate", "Government Spending", "Output")
sim_seq_l_plot = hcat(sim_seq_l[1:3]..., sim_seq_l[4],
        time_example.G[sHist_l],
        time_example.Θ[sHist_l] .* sim_seq_l[2])
sim_bel_l_plot = hcat(sim_bel_l[1:3]..., sim_bel_l[5],
        time_example.G[sHist_l],
```

6.4. Optimal Taxation without State-Contingent Debt
How a Ramsey planner responds to war depends on the structure of the asset market.
If it is able to trade state-contingent debt, then at time $t = 2$
• the government purchases an Arrow security that pays off when $g_3 = g_h$.
• the government sells an Arrow security that pays off when $g_3 = g_l$.
• These purchases are designed in such a way that regardless of whether or not there is a war at $t = 3$, the government will begin period $t = 4$ with the same government debt.

This pattern facilitates smoothing tax rates across states.

The government without state contingent debt cannot do this.

Instead, it must enter time $t = 3$ with the same level of debt falling due whether there is peace or war at $t = 3$.

It responds to this constraint by smoothing tax rates across time.

To finance a war it raises taxes and issues more debt.

To service the additional debt burden, it raises taxes in all future periods.

The absence of state contingent debt leads to an important difference in the optimal tax policy.

When the Ramsey planner has access to state contingent debt, the optimal tax policy is history independent.

• the tax rate is a function of the current level of government spending only, given the Lagrange multiplier on the implementability constraint.

Without state contingent debt, the optimal tax rate is history dependent.

• A war at time $t = 3$ causes a permanent increase in the tax rate.

Perpetual War Alert

History dependence occurs more dramatically in a case in which the government perpetually faces the prospect of war.

This case was studied in the final example of the lecture on optimal taxation with state-contingent debt.

There, each period the government faces a constant probability, 0.5, of war.

In addition, this example features the following preferences

$$u(c, n) = \log(c) + 0.69 \log(1 - n)$$

In accordance, we will re-define our utility function:

```julia
function log_utility(; β = 0.9,
    ψ = 0.69,
    Π = 0.5 * ones(2, 2),
    G = [0.1, 0.2],
    Θ = ones(2),
    transfers = false)

    # Derivatives of utility function
    U(c,n) = log(c) + ψ * log(1 - n)
    Uc(c,n) = 1 ./ c
    Ucc(c,n) = -c.^(-2.0)
```
With these preferences, Ramsey tax rates will vary even in the Lucas-Stokey model with state-contingent debt.

The figure below plots optimal tax policies for both the economy with state contingent debt (circles) and the economy with only a risk-free bond (triangles).

```julia
log_example = log_utility()
log_example.transfers = true               # Government can use transfers
log_sequential = SequentialAllocation(log_example)  # Solve sequential problem
log_bellman = RecursiveAllocation(log_example, μgrid)  # Solve recursive problem

T = 20
sHist = [1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 2, 2, 2, 2, 2, 2, 1]

# simulate
sim_seq = simulate(log_sequential, 0.5, 1, T, sHist)
sim_bel = simulate(log_bellman, 0.5, 1, T, sHist)

sim_seq_plot = hcat(sim_seq[1:3],
                     sim_seq[4], log_example.G[sHist], log_example.Θ[sHist] .* sim_
                     seq[2])
sim_bel_plot = hcat(sim_bel[1:3],
                     sim_bel[5], log_example.G[sHist], log_example.Θ[sHist] .* sim_
                     bel[2])

# plot policies
p = plot(size = (700, 500), layout = grid(3, 2),
          xaxis=(0:T), grid=false, titlefont=Plots.font("sans-serif", 10))
labels = fill(("", ""), 6)
labels[1] = ("Complete Market", "Incomplete Market")
plot!(p, title = titles)
for i = vcat(collect(1:4), 6)
    plot!(p[i], sim_seq_plot[:, i], marker=:circle, color=:blue, lab=labels[i][1])
    plot!(p[i], sim_bel_plot[:, i], marker=:utriangle, color=:black, lab=labels[i][1])
end
plot!(p[5], sim_seq_plot[:, 5], marker=:circle, color=:black, lab="")```
When the government experiences a prolonged period of peace, it is able to reduce government debt and set permanently lower tax rates.

However, the government finances a long war by borrowing and raising taxes.

This results in a drift away from policies with state contingent debt that depends on the history of shocks.

This is even more evident in the following figure that plots the evolution of the two policies over 200 periods.
plot!(p[i], sim_bel_long_plot[:, i], color=blue, linestyle=dot,
    lab=labels[i][2])
end
CHAPTER
SEVEN

REFERENCES
This section of the course contains foundational mathematical and statistical tools and techniques.
This section of the course contains foundational models for dynamic economic modeling. Most are single agent problems that take the activities of other agents as given. Later we will look at full equilibrium problems.
MULTIPLE AGENT MODELS

These lectures look at important economic models that also illustrate common equilibrium concepts.

Acknowledgements: These lectures have benefitted greatly from comments and suggestion from our colleagues, students and friends. Special thanks go to Anmol Bhandari, Long Bui, Jeong-Hun Choi, Chase Coleman, David Evans, Shunsuke Hori, Chenghan Hou, Doc-Jin Jang, Spencer Lyon, Qingyin Ma, Akira Matsushita, Matthew McKay, Tomohito Okabe, Alex Olssen, Nathan Palmer and Yixiao Zhou.
This section of the course contains foundational mathematical and statistical tools and techniques
This section of the course contains foundational models for dynamic economic modeling. Most are single agent problems that take the activities of other agents as given. Later we will look at full equilibrium problems.
MULTIPLE AGENT MODELS

These lectures look at important economic models that also illustrate common equilibrium concepts.
These lectures look at important concepts in time series that are used in economics.

Acknowledgements: These lectures have benefitted greatly from comments and suggestion from our colleagues, students and friends. Special thanks go to Anmol Bhandari, Long Bui, Jeong-Hun Choi, Chase Coleman, David Evans, Shunsuke Hori, Chenghan Hou, Doc-Jin Jang, Spencer Lyon, Qingyin Ma, Akira Matsushita, Matthew McKay, Tomohito Okabe, Alex Olssen, Nathan Palmer and Yixiao Zhou.


<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s)</th>
<th>Title</th>
<th>Publisher, Edition</th>
</tr>
</thead>
</table>